EE414 Embedded Systems

Ch 10. Operating Systems

Part 1/2

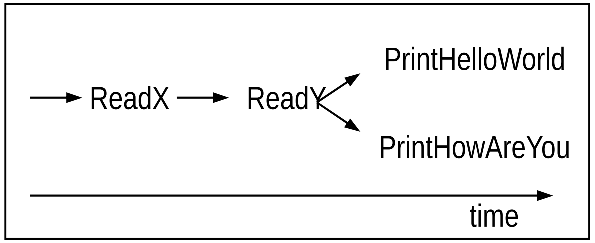
Byung Kook Kim
School of Electrical Engineering
Korea Advanced Institute of Science and Technology

Overview

- 10.1 Concurrent Process Model
- 10.2 Concurrent Processes
- 10.3 Communication Among Processes
- 10.4 Synchronization Among Processes
- 10.5 Implementation
- 10.6 Operating Systems

10.1 Concurrent Process Model

- **Concurrent Process Model**
  - A model describing functionality of system in terms of *two or more concurrently executing subtasks*
  - Many systems easier to describe with concurrent process model because inherently *multitasking*
- E.g., simple example:
  - Read two numbers *X* and *Y*
    - Display “Hello world.” every *X* seconds
    - Display “How are you?” every *Y* seconds →
- *More effort* would be required with sequential program or state machine model.



Subroutine execution over time

Enter X: *1*
Enter Y: *2*
Hello world. *(Time = 1 s)*
Hello world. *(Time = 2 s)*
How are you? *(Time = 2 s)*
Hello world. *(Time = 3 s)*
How are you? *(Time = 4 s)*
Hello world. *(Time = 4 s)*
...

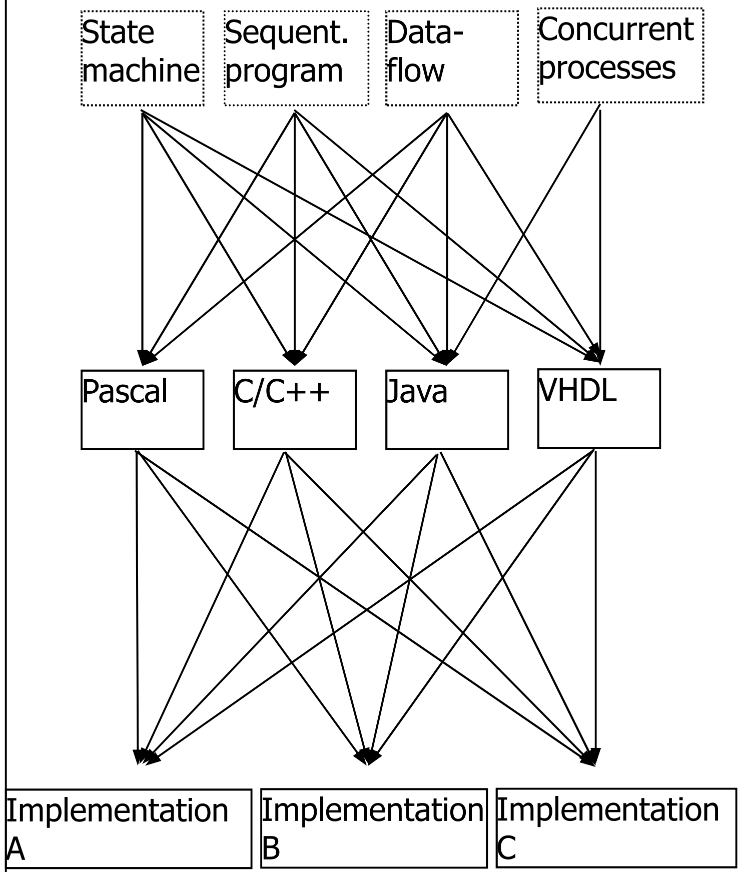
Sample input and output

```
ConcurrentProcessExample() {
  x = ReadX()
  y = ReadY()
  Call concurrently:
    PrintHelloWorld(x) and
    PrintHowAreYou(y)
}

PrintHelloWorld(x) {
  while( 1 ) {
    print "Hello world."
    delay(x);
  }
}

PrintHowAreYou(x) {
  while( 1 ) {
    print "How are you?"
    delay(y);
  }
}
```

Simple concurrent process example

Implementation

- Mapping of system's functionality onto hardware processors:
  - captured using computational model(s)
  - written in some language(s)
- Implementation choice independent from language(s) choice
- Implementation choice based on power, size, performance, timing and cost requirements
- Final implementation tested for feasibility
  - Also serves as blueprint/prototype for mass manufacturing of final product



The choice of computational model(s) is based on whether it allows the designer to describe the system.

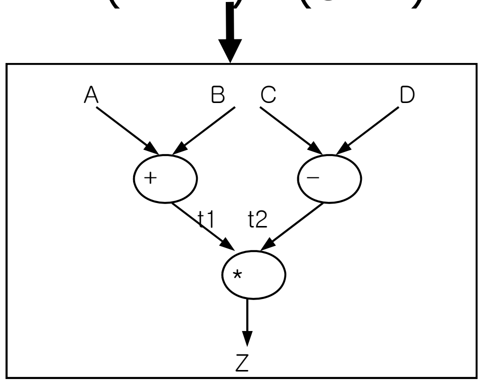
The choice of language(s) is based on whether it captures the computational model(s) used by the designer.

The choice of implementation is based on whether it meets power, size, performance and cost requirements.

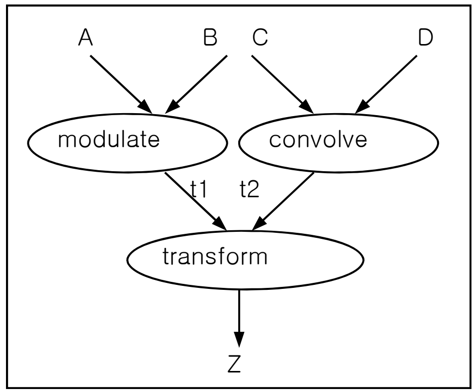
Dataflow model

- Derivative of concurrent process model
- Nodes represent transformations
  - May execute concurrently
- Edges represent flow of tokens (data) from one node to another
  - May or may not have token at any given time
- When all of node's input edges have at least one token, node may fire
- When node fires, it consumes input tokens, processes transformation, and generates output token
- Nodes may fire simultaneously
- Several commercial tools support graphical languages for capture of dataflow model
  - Can automatically translate to concurrent process model for implementation
  - Each node becomes a process

Z = (A + B) * (C - D)



Nodes with arithmetic transformations

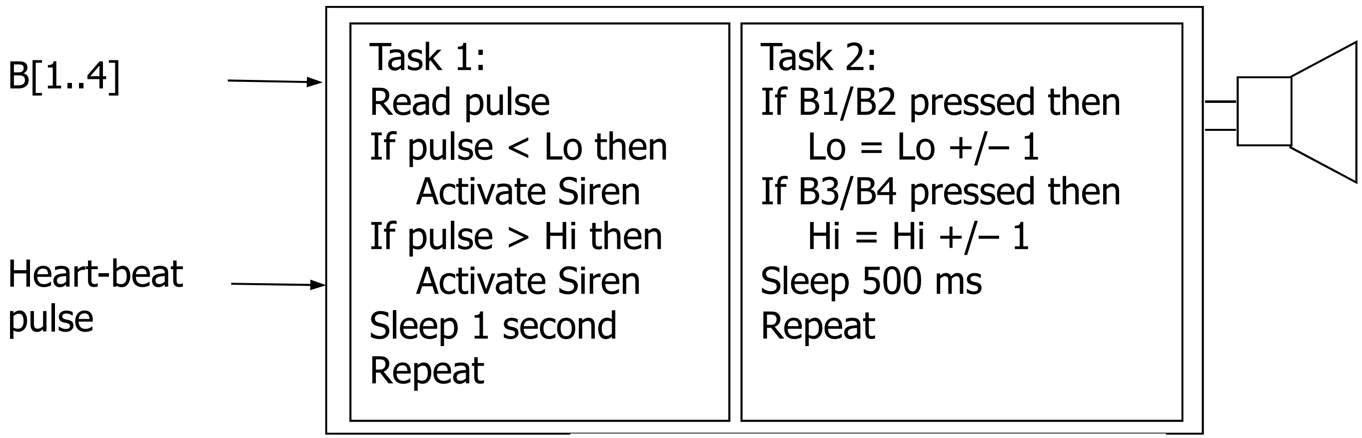


Nodes with more complex transformations

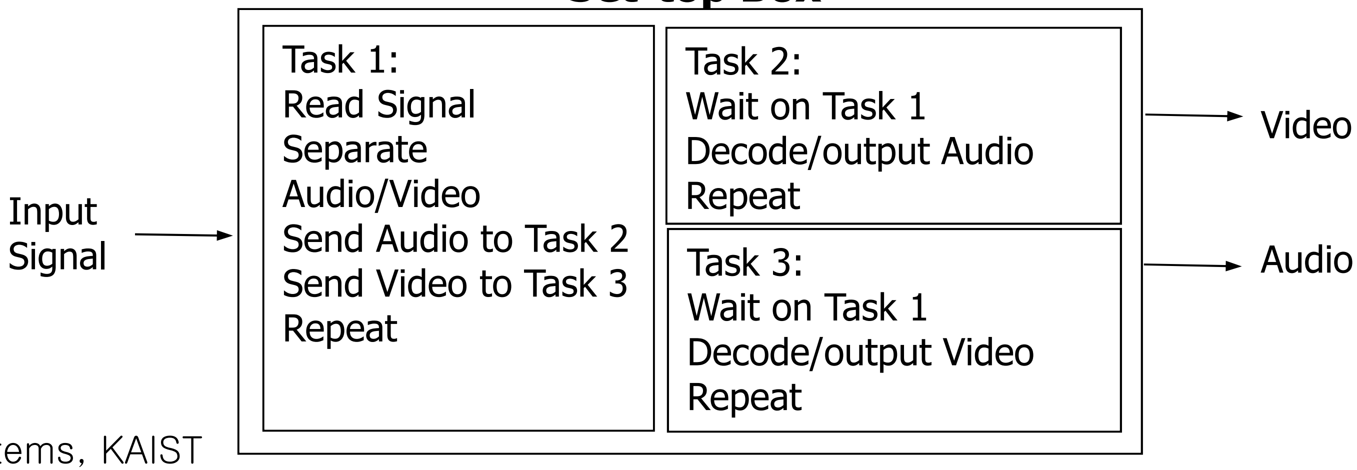
10.2 Concurrent Processes

- Consider two examples having separate tasks running independently but sharing data

Heartbeat Monitoring System



Set-top Box



Concurrent Processes (II)

- Difficult to write system using sequential program model.
- Concurrent process model easier.
  - Separate sequential programs (processes) for each task
  - Programs communicate with each other.
- The operating system manages processes.

Process

- A sequential program, typically an infinite loop, among concurrent processes.
- A unit of execution: Executes concurrently with other processes
  - We are about to enter the world of "concurrent programming"
- Organize executable code into manageable units

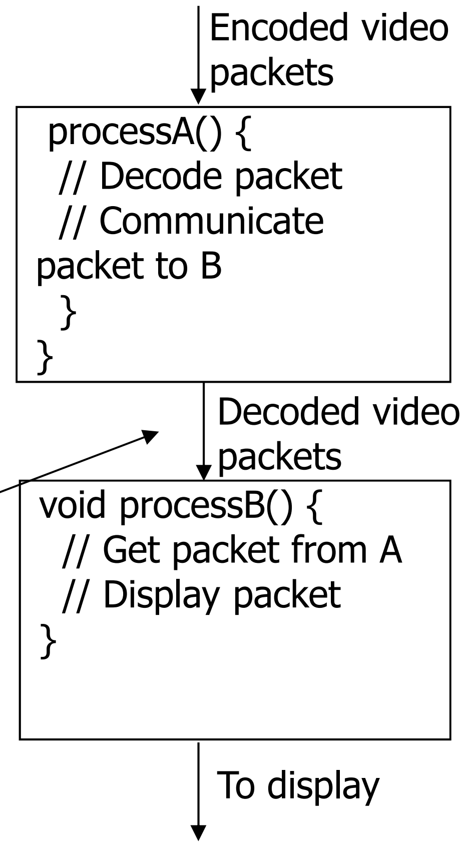
- **Process's state**
  - Running: Currently being executed.
  - Runnable: Ready and executable.
  - Blocked: Not ready to be executed (wait for other process, device operation).

Process Operations

- Create and terminate
  - Create: Creates a new process, initialize data, and start execution.
    - Asynchronous procedure call but caller doesn't wait (blocked)
    - Created process can itself create new processes, start executing concurrently.
  - Terminate: Termiates already executing process, destroying all associated data.
- Suspend and resume
  - Suspend: puts a process on hold, saving state for later execution.
  - Resume: starts the process again where it left off.
- Join
  - Join: A process suspends until another process finishes execution.
    - Synchronization of processes.

10.3 Communication Among Processes

- Processes need to communicate data and signals to solve their computation problem.
  - Processes that don't communicate are just independent programs solving separate problems.
- Basic example: producer/consumer
  - Process A produces data items, Process B consumes them
  - E.g., A decodes video packets, B display decoded packets on a screen
- How do we achieve this communication?
  - Two basic methods
    - Shared memory
    - Message passing



A. Shared Memory

- Processes read and write shared variables
  - No time overhead, easy to implement
  - But hard to use – mistakes are common
- Example: Producer/consumer with a mistake
  - Share *buffer*[*N*], *count*
    - *count* = # of valid data items in *buffer*
  - *processA* produces data items and stores in *buffer*
    - If *buffer* is full, must wait
  - *processB* consumes data items from *buffer*
    - If *buffer* is empty, must wait
  - Error when both processes try to update *count* concurrently (lines 10 and 19) and the following execution sequence occurs. Say "count" is 3.
    - *A* loads *count* (*count* = 3) from memory into register R1 (R1 = 3)
    - *A* increments R1 (R1 = 4)
    - *B* loads *count* (*count* = 3) from memory into register R2 (R2 = 3)
    - *B* decrements R2 (R2 = 2)
    - *A* stores R1 back to *count* in memory (*count* = 4)
    - *B* stores R2 back to *count* in memory (*count* = 2)
  - *count* now has incorrect value of 2!

```
01: data_type buffer[N];
02: int count = 0;
03: void processA() {
04:   int i;
05:   while( 1 ) {
06:     produce(&data);
07:     while( count == N
);/*loop*/
08:     buffer[i] = data;
09:     i = (i + 1) % N;
10:     count = count + 1;
11:   }
12: }
13: void processB() {
14:   int i;
15:   while( 1 ) {
16:     while( count == 0
);/*loop*/
17:     data = buffer[i];
18:     i = (i + 1) % N;
19:     count = count - 1;
20:     consume(&data);
21:   }
22: }
23: void main() {
24:   create_process(processA);
25:   create_process(processB);
26: }
```

Shared Memory: Mutual Exclusion

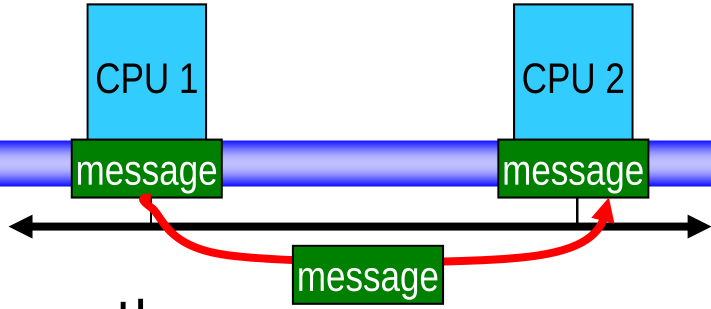
- Certain sections of code should not be performed concurrently
  - **Critical section**
    - Possibly noncontiguous section of code where simultaneous updates, by multiple processes to a shared memory location, can occur
- When a process enters the critical section, all other processes must be locked out until it leaves the critical section
- **Mutex**
  - A shared object used for locking and unlocking segment of shared data
  - Disallows read/write access to memory it guards
  - Multiple processes can perform lock operation simultaneously, but only one process will acquire lock
  - All other processes trying to obtain lock will be put in blocked state until unlock operation performed by acquiring process when it exits critical section
  - These processes will then be placed in runnable state and will compete for lock again

Correct Shared Memory Solution to the Consumer-Producer Problem

- The primitive *mutex* is used to ensure critical sections are executed in mutual exclusion of each other
- Following the same execution sequence as before:
  - *A*/*B* execute *lock* operation on *count_mutex*
  - Either *A* **or** *B* will acquire *lock*
    - Say *B* acquires it
    - *A* will be put in blocked state
  - *B* loads *count* (*count* = 3) from memory into register R2 (R2 = 3)
  - *B* decrements R2 (R2 = 2)
  - *B* stores R2 back to *count* in memory (*count* = 2)
  - *B* executes *unlock* operation
    - *A* is placed in runnable state again
  - *A* loads *count* (*count* = 2) from memory into register R1 (R1 = 2)
  - *A* increments R1 (R1 = 3)
  - *A* stores R1 back to *count* in memory (*count* = 3)
- *Count* now has correct value of 3!

```
data_type buffer[N];
int count = 0;
mutex count_mutex;
void processA() {
  int i;
  while( 1 ) {
    produce(&data);
    while( count == N );/*loop*/
    buffer[i] = data;
    i = (i + 1) % N;
    count_mutex.lock();
    count = count + 1;
    count_mutex.unlock();
  }
}
void processB() {
  int i;
  while( 1 ) {
    while( count == 0 );/*loop*/
    data = buffer[i];
    i = (i + 1) % N;
    count_mutex.lock();
    count = count - 1;
    count_mutex.unlock();
    consume(&data);
  }
}
void main() {
  create_process(processA);
  create_process(processB);
}
```

B. Message Passing



- **Message passing**
  - Data explicitly sent from one process to another
    - Sending process performs special operation, *send*
    - Receiving process must perform special operation, *receive*, to receive the data
    - Both operations must explicitly specify which process it is sending to or receiving from
    - Receive is *blocking*, send *may or may not be blocking*
  - Safer model, but less flexible

```
void processA() {
  while( 1 ) {
    produce(&data)
    send(B, &data);
    /* region 1 */
    receive(B, &data);
    consume(&data);
  }
}
```

```
void processB() {
  while( 1 ) {
    receive(A, &data);
    transform(&data)
    send(A, &data);
    /* region 2 */
  }
}
```

10.4 Synchronization Among Processes

- Sometimes concurrently running processes must synchronize their execution
  - When a process must wait for:
    - another process to compute some value
    - reach a known point in their execution
    - signal some condition
- Recall producer-consumer problem
  - *processA* must wait if *buffer* is full
  - *processB* must wait if *buffer* is empty
  - This is called busy-waiting
    - Process executing loops instead of being blocked
    - CPU time wasted
- More efficient methods
  - Join operation, and blocking send and receive discussed earlier
    - Both block the process so it doesn’t waste CPU time
  - Condition variables and monitors.

A. Condition variables

- Condition variable is an object that has 2 operations, signal and wait
- When process performs a wait on a condition variable, the process is blocked until another process performs a signal on the same condition variable
- How is this done?
  - Process *A* acquires lock on a mutex
  - Process *A* performs wait, passing this mutex
    - Causes mutex to be unlocked
  - Process *B* can now acquire lock on same mutex
  - Process *B* enters critical section
    - Computes some value and/or make condition true
  - Process *B* performs signal when condition true
    - Causes process *A* to implicitly reacquire mutex lock
    - Process *A* becomes runnable

Condition variable example: consumer-producer

- 2 condition variables
  - *buffer_empty*
    - Signals at least 1 free location available in *buffer*
  - *buffer_full*
    - Signals at least 1 valid data item in *buffer*
- *processA*:
  - produces data item
  - acquires lock (*cs_mutex*) for critical section
  - checks value of *count*
  - if *count = N*, *buffer* is full
    - performs wait operation on *buffer_empty*
    - this releases the lock on *cs_mutex* allowing *processB* to enter critical section, consume data item and free location in *buffer*
    - *processB* then performs signal
  - if *count* < *N*, *buffer* is not full
    - *processA* inserts data into *buffer*
    - increments *count*
    - signals *processB* making it runnable if it has performed a wait operation on *buffer_full*

Consumer−producer using condition variables

```
data_type buffer[N];
int count = 0;
mutex cs_mutex;
condition buffer_empty, buffer_full;
void processA() {
  int i;
  while( 1 ) {
    produce(&data);
    cs_mutex.lock();
    if( count == N ) buffer_empty.wait(cs_mutex);
    buffer[i] = data;
    i = (i + 1) % N;
    count = count + 1;
    cs_mutex.unlock();
    buffer_full.signal();
  }
}
void processB() {
  int i;
  while( 1 ) {
    cs_mutex.lock();
    if( count == 0 ) buffer_full.wait(cs_mutex);
    data = buffer[i];
    i = (i + 1) % N;
    count = count - 1;
    cs_mutex.unlock();
    buffer_empty.signal();
    consume(&data);
  }
}
void main() {
  create_process(processA); create_process(processB);
}
```

B. Monitors

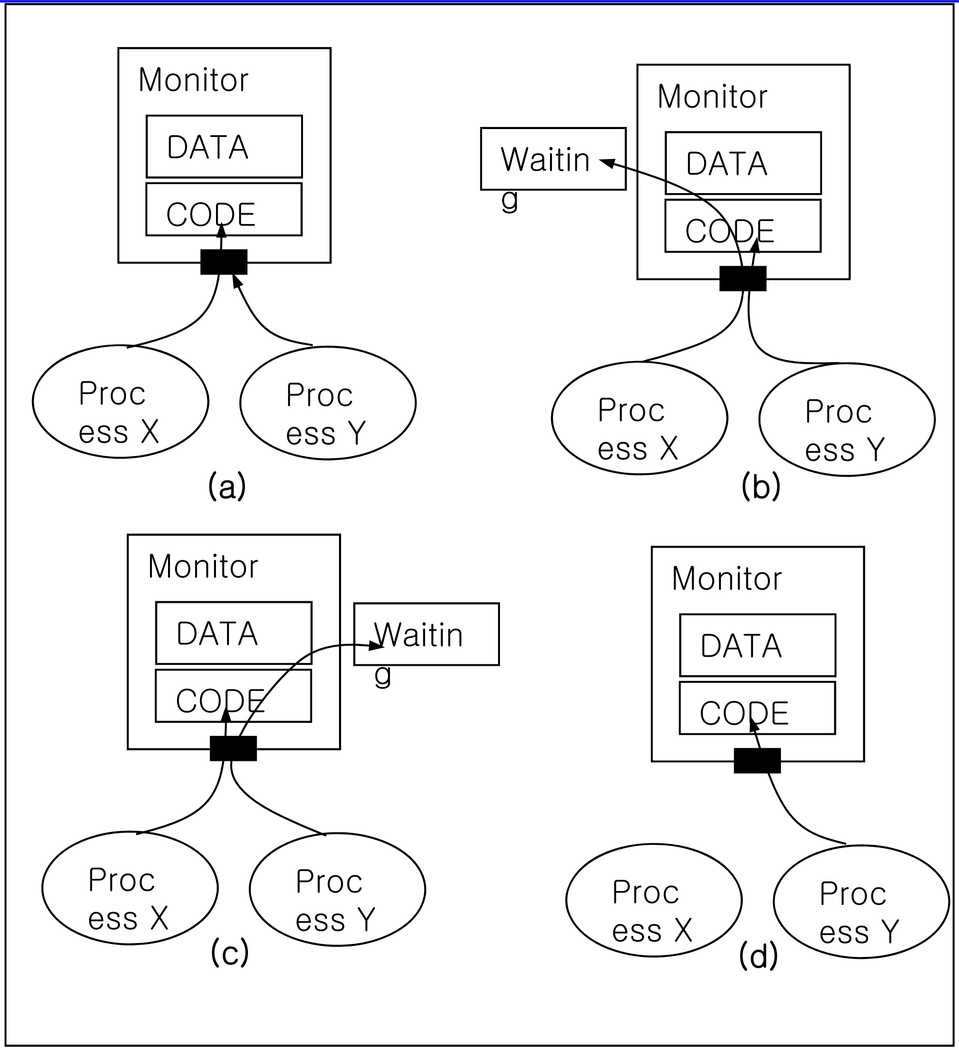
- Collection of data and methods or subroutines that operate on data similar to an object-oriented paradigm
- Guarding: guarantees only 1 process can execute inside monitor at a time

- (a) Process X executes while Process Y has to wait

- (b) Process X performs wait on a condition
  - Process Y allowed to enter and execute

- (c) Process Y signals condition Process X waiting on
  - Process Y blocked
  - Process X allowed to continue executing

- (d) Process X finishes executing in monitor or waits on a condition again
  - Process Y made runnable again



Monitor example: consumer-producer

- Single monitor encapsulates both processes along with *buffer* and *count*
- One process will be allowed to begin executing first

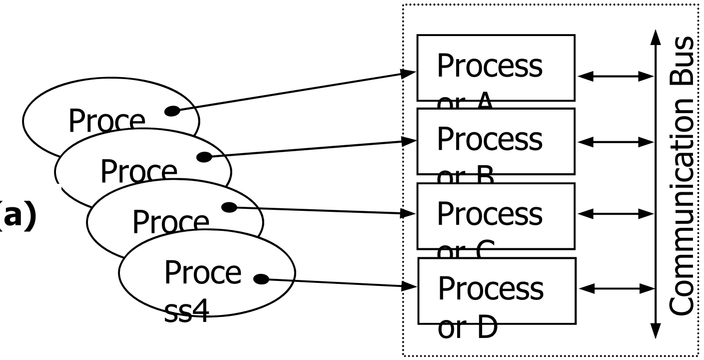
- If *processB* allowed to execute first
  - Will execute until it finds *count* = 0
  - Will perform wait on *buffer_full* condition variable
  - *processA* now allowed to enter monitor and execute
  - *processA* produces data item
  - finds *count* < *N* so writes to *buffer* and increments *count*
  - *processA* performs signal on *buffer_full* condition variable
  - *processA* blocked
  - *processB* reenters monitor and continues execution, consumes data, etc.

```
Monitor {
  data_type buffer[N];
  int count = 0;
  condition buffer_full, condition buffer_empty;
  void processA() {
    int i;
    while( 1 ) {
      produce(&data);
      if( count == N ) buffer_empty.wait();
      buffer[i] = data;
      i = (i + 1) % N;
      count = count + 1;
      buffer_full.signal();
    }
  }
  void processB() {
    int i;
    while( 1 ) {
      if( count == 0 ) buffer_full.wait();
      data = buffer[i];
      i = (i + 1) % N;
      count = count - 1;
      buffer_empty.signal();
      consume(&data);
      buffer_full.signal();
    }
  }
}  /* end monitor */
void main() {
  create_process(processA); create_process(processB);
}
```

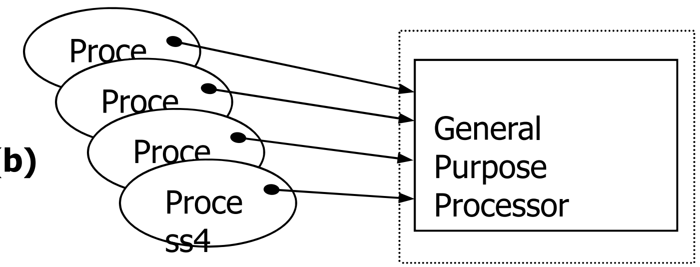
10.5 Implementation

- Can use single and/or general-purpose processors
- (a) Multiple processors, each executing one process
  - True multitasking (parallel processing)
  - General-purpose processors
    - Use programming language like C and compile to instructions of processor
    - Expensive and in most cases not necessary
  - Custom single-purpose processors
    - More common
- (b) One general-purpose processor running all processes
  - Most processes don't use 100% of processor time
  - Can share processor time and still achieve necessary execution rates
- (c) Combination of (a) and (b)
  - Multiple processes run on one general-purpose processor while one or more processes run on own single_purpose processor

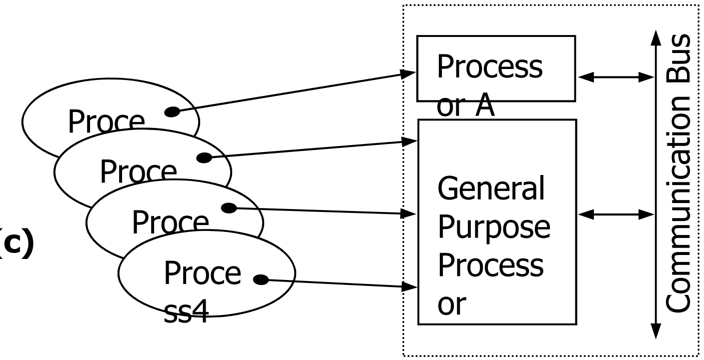
(a)



(b)



(c)



Implementation: Multiple processes sharing single processor

- Can manually rewrite processes as a single sequential program
  - Ok for simple examples, but extremely difficult for complex examples
  - Automated techniques have evolved but not common
- Can convert processes to sequential program with process scheduling right in code
  - Less overhead (no operating system)
  - More complex/harder to maintain
- Can use multitasking operating system
  - Much more common
  - Operating system schedules processes, allocates storage, and interfaces to peripherals, etc.
  - Real-time operating system (RTOS) can guarantee execution rate constraints are met
  - Describe concurrent processes with languages having built-in processes (Java, Ada, etc.) or a sequential programming language with library support for concurrent processes (C, C++, etc. using POSIX threads for example)

Processes vs. threads

- Different meanings when operating system terminology
- Processes
  - Heavyweight process
  - Own virtual address space (stack, data, code)
  - System resources (e.g., open files)
- Threads
  - Lightweight process
  - Subprocess within process
  - Only program counter, stack, and registers
  - Shares address space, system resources with other threads
    - Allows quicker communication between threads
    - May destroy data of other thread.
  - Small compared to heavyweight processes
    - Can be created quickly
    - Low cost switching between threads

Implementation: suspending, resuming, and joining

- Multiple processes mapped to single-purpose processors
  - Built into processor's implementation
  - Could be extra input signal that is asserted when process suspended
  - Additional logic needed for determining process completion
    - Extra output signals indicating process done

- Multiple processes mapped to single general-purpose processor
  - Built into programming language or special multitasking library like POSIX
  - Language or library may rely on operating system to handle

Implementation: process scheduling

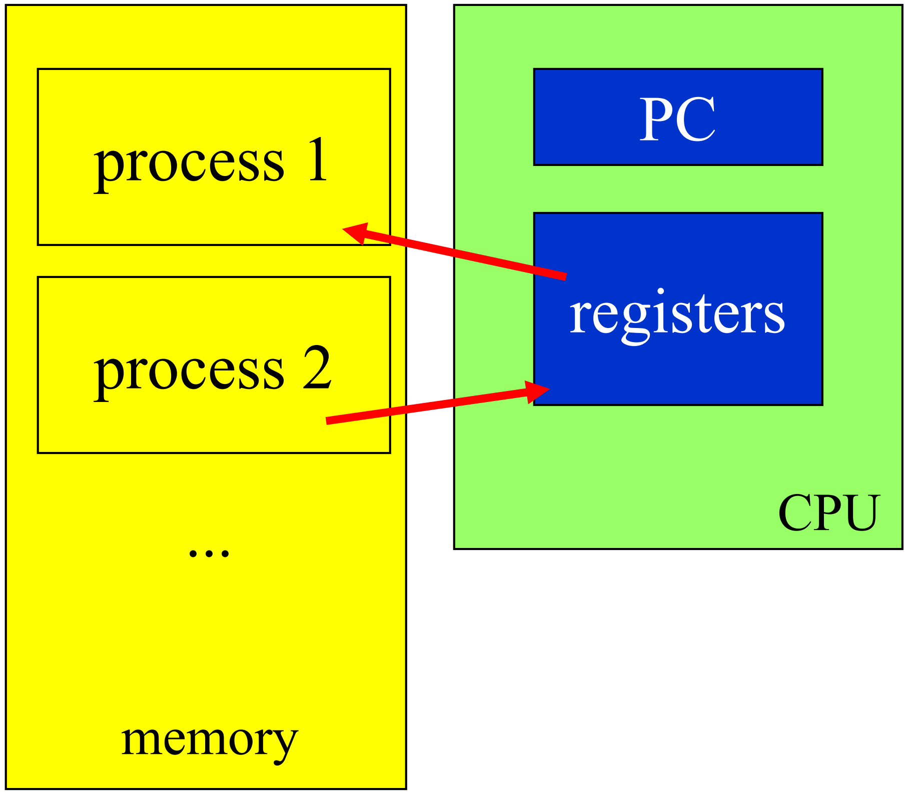
- Must meet timing requirements when multiple concurrent processes implemented on single general-purpose processor
  - Not true multitasking
- Scheduler
  - Special process that decides when and for how long each process is executed
  - Implemented as preemptive or nonpreemptive scheduler
  - Preemptive
    - Determines how long a process executes before preempting to allow another process to execute
      - Time quantum: predetermined amount of execution time preemptive scheduler allows each process (may be 10 to 100s of milliseconds long)
    - Determines which process will be next to run
  - Nonpreemptive
    - Only determines which process is next after current process finishes execution

Scheduling: priority

- Process with highest priority always selected first by scheduler
  - Typically determined statically during creation and dynamically during execution
- FIFO
  - Runnable processes added to end of FIFO as created or become runnable
  - Front process removed from FIFO when time quantum of current process is up or process is blocked
- Priority queue
  - Runnable processes again added as created or become runnable
  - Process with highest priority chosen when new process needed
  - If multiple processes with same highest priority value then selects from them using first-come first-served
  - Called priority scheduling when nonpreemptive
  - Called round-robin when preemptive

Context Switching

- Process activation record:
  - Copy of process state.
  - Data used to reactivate the process.
- Context switch:
  - Current CPU context goes out;
  - New CPU context goes in.



Context Switching

- **Context switch**
  - Mechanism for moving the CPU from one executing process to another.
  - Must be bug free and fast.

- Questions
  - *Who* controls when the context is switched?
  - *How* is the context switched?

10.6 Operating Systems

- The operating system controls resources:
  - who gets the CPU;
  - when I/O takes place;
  - how much memory is allocated.
- The most important (scarcest) resource is the CPU itself.
  - *CPU access* controlled by the scheduler.

Embedded vs. General-Purpose Scheduling

- General-purpose scheduling
  - Workstations try to avoid starving processes of CPU access.
    - Fairness = access to CPU.
- Embedded scheduling
  - Embedded systems must meet deadlines
  - Low-priority processes may not run for a long time.

Operating System Structure

- OS needs to keep track of process activation record:
  - process priorities;
  - process scheduling state;
  - Starting address of the process.
- Processes may be created:
  - statically before system starts – array
  - dynamically during execution – linked list.
- The operating system generally execute in protected mode.

Other Operating System Functions

- Managing shared resources
  - CPU, devices
- Driver
  - I/O devices
  - Networking
- Date/time
- File system
- Security.

References

- [1] Frank Vahid, “Embedded system design: A unified hardware/software introduction”, John Wiley & Sons, 2002.
- [2] Wayne Wolf, “Computers as Components”, Morgan Kaufman, 2001.