Operating Systems: Inter-process Communication

Processes

- Basic concept to build the OS around
  - From old IBM mainframe OS to the most modern Windows
  - **Fundamental task of OS: Process management**
- Used to express requirements that OS needs to meet
  - Interleave execution of multiple processes, maximize CPU utilization while providing good response time
  - Allocate resources to processes using a policy while avoiding deadlocks
  - Support IPC and user creation of processes to help structuring applications

Background

- Computer platform
  - Collection of hardware resources - CPU, memory, I/O modules, timers, storage devices
- Computer applications
  - Developed to perform some task
  - Input, processing, output
- Efficient to write applications for a given CPU
  - Common routines to access computer resources across platforms
  - CPU only provides limited support for multiprogramming, software needs to do most
  - Protect resources from other applications

What is a process?

- A program in execution
- Abstraction of a running program
- Entity that can be assigned to and executed on a processor
- Unit of work in a system, characterized by:
  - Execution of a sequence of instructions
  - A current state
  - Associated set of machine instructions

Processes

- Split into two abstractions in modern OS
  - Resources ownership (traditional process view)
  - Stream of instruction execution (thread)
- Pseudoparallelism, or interleaved instructions
- A process is traced by listing the sequence of instructions that execute for that process

Process elements

- A process is comprised of:
  - Program code (possibly shared)
  - A set of data
  - A number of attributes describing the state of the process

Process elements

- While the process is running, it has a number of elements:
  - Identifier
  - State
  - Priority
  - Program counter
  - Memory pointers
  - Context data (registers, etc)
  - I/O status information
  - Accounting information

Process Control Block

- Contains process elements
- Created and managed by OS
- Allows support for multiple processes

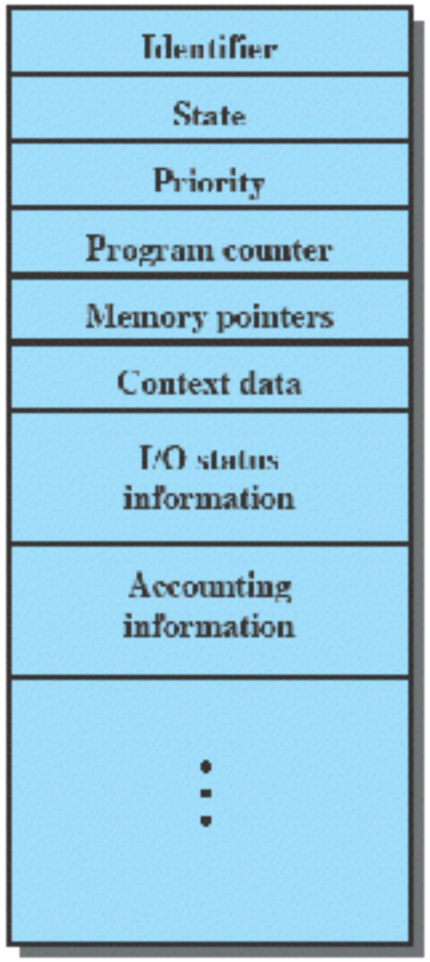


Figure 3.1 Simplified Process Control Block

Trace of the process

- The behavior of an individual process is shown by listing the sequence of instructions that are executed
  - This list is called a *Trace*
- Dispatcher is a small program which switches the processor from one process to another

Concurrent Processes

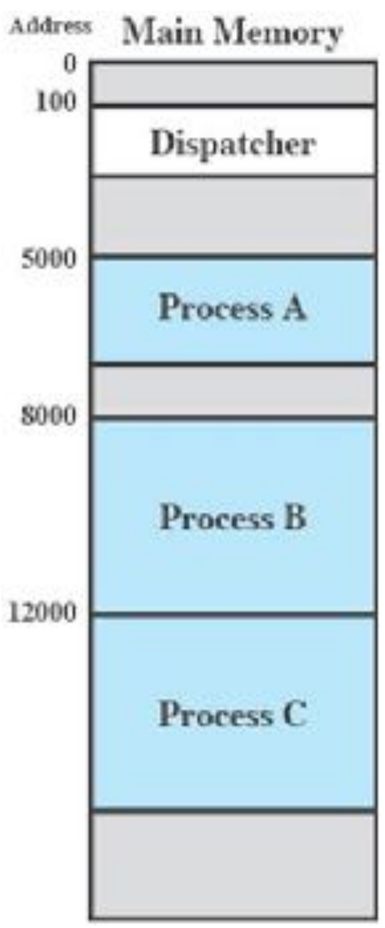
- Multiprogramming
- Interleaving of traces of different processes characterizes the behavior of the CPU
- Physical resource sharing
  - Required due to limited hardware resources
- Logical resource sharing
  - Concurrent access to the same resource like files

Concurrent Processes

- Computation speedup
  - Break each task into subtasks
  - Execute each subtask on separate processing element
- Modularity
  - Division of system functions into separate modules
- Convenience
  - Perform a number of tasks in parallel
- Real-time requirements for I/O

Process Execution

- Consider three processes being executed
- All are in memory (plus dispatcher)
- No virtual memory atm



Trace from processes pov

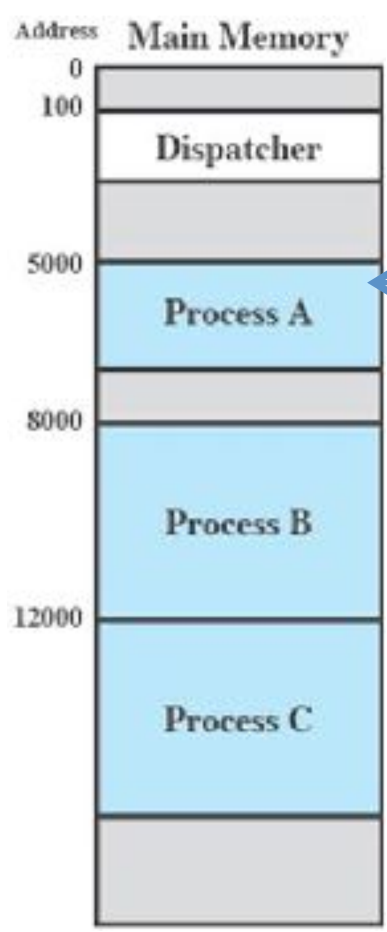
- Each process runs to completion

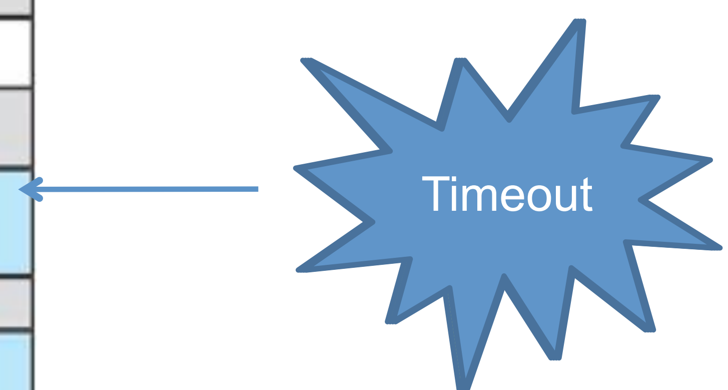
(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.3 Traces of Processes of Figure 3.2

Trace from processes pov





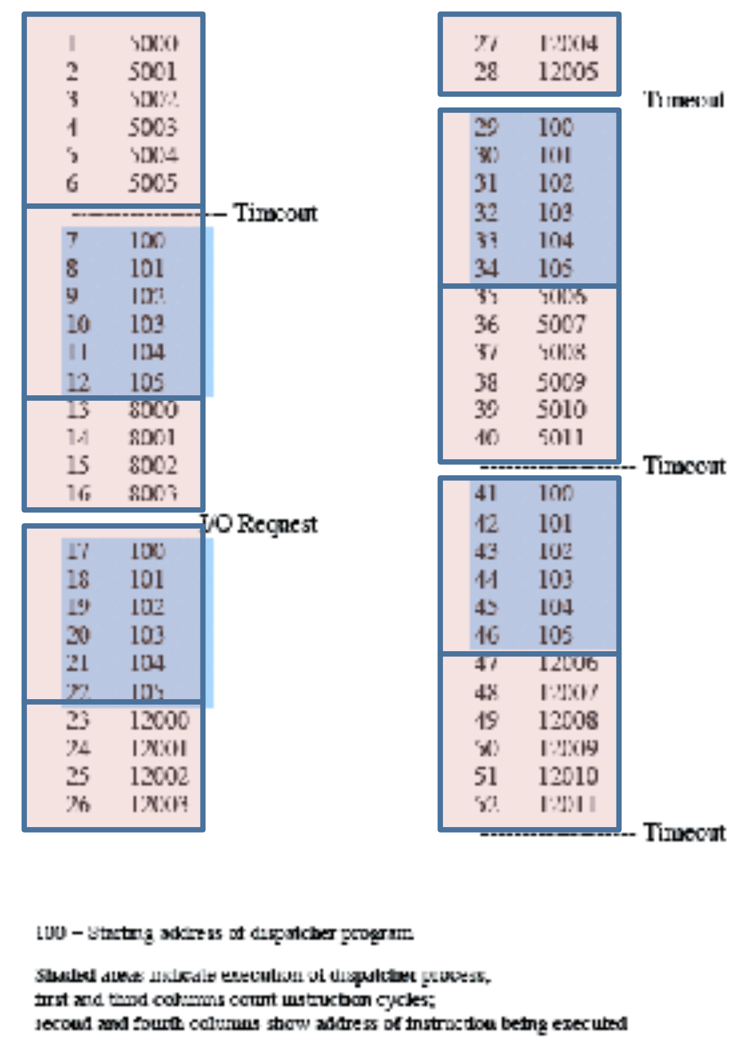


Figure 3.4 Combined Trace of Processes of Figure 3.2

Process Hierarchies

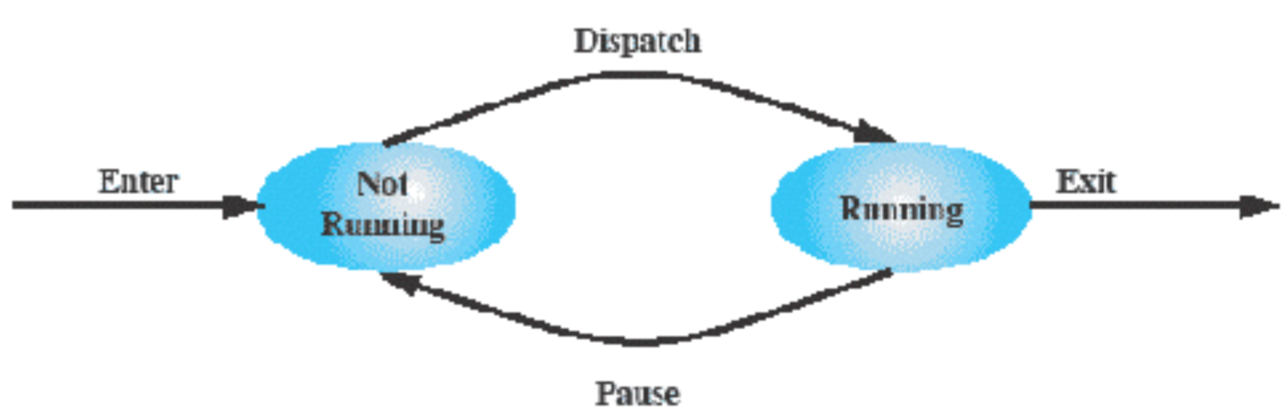
- Parent-child relationship
  - fork system call in Unix
  - In older non-multitasking systems such as MS-DOS, parent suspends itself and lets child execute

Process states

- Main topics are:
  - How are processes represented and controlled by the OS?
  - Process states which characterize the behavior of processes
  - Data structures used to manage processes
  - Ways in which the OS uses these data structures to control process execution

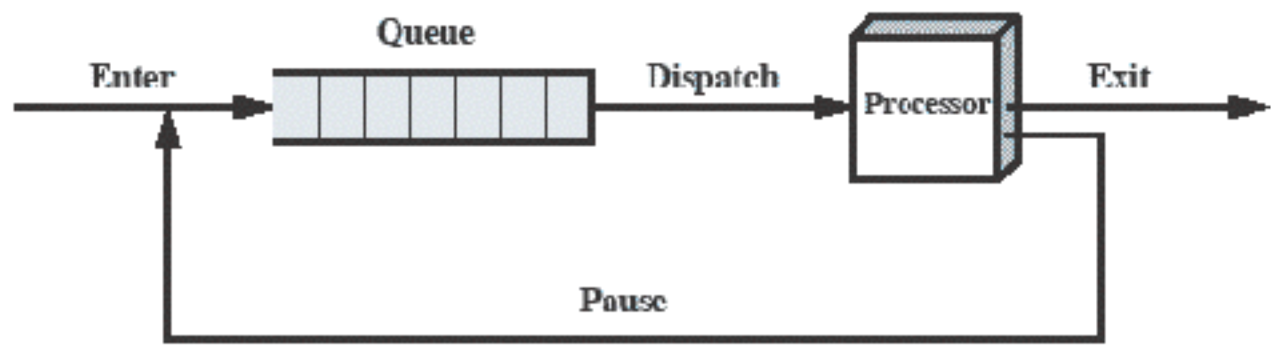
Two-state process Model

- Simplest possible model
- A process is either executing (running state) or it is idle (not running)
- For new process, OS creates a new process control block and brings that process into memory
    - Initially in a non-running state



(a) State transition diagram

Queuing Diagram



(b) Queuing diagram

- Processes moved by dispatcher of the OS to the CPU and then back to the queue until the task is completed.

Two-state model

- Problems with 2-state model?
  - First, what if many processes NotRunning might be ready to run, while others NotRunning are blocked?
    - Could split NotRunning into two states
      - Blocked and Ready

Five-state process Model

- Five-state model
  - Running - Process currently executing
  - Ready - Not running, waiting for the CPU
  - Blocked - Waiting on an event (other than the CPU)
  - Two other states to fill it out
    - New - A process being created and will be Ready when done
    - Exit - A process being terminated

Five-state process Model

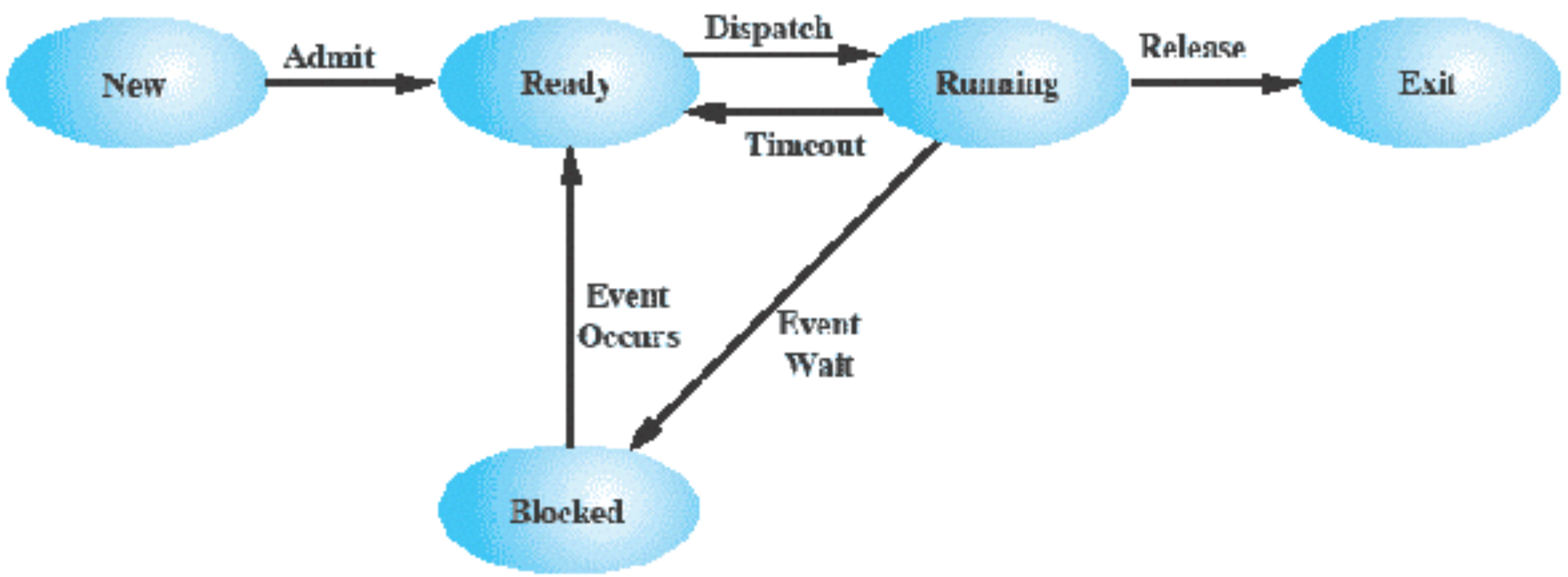
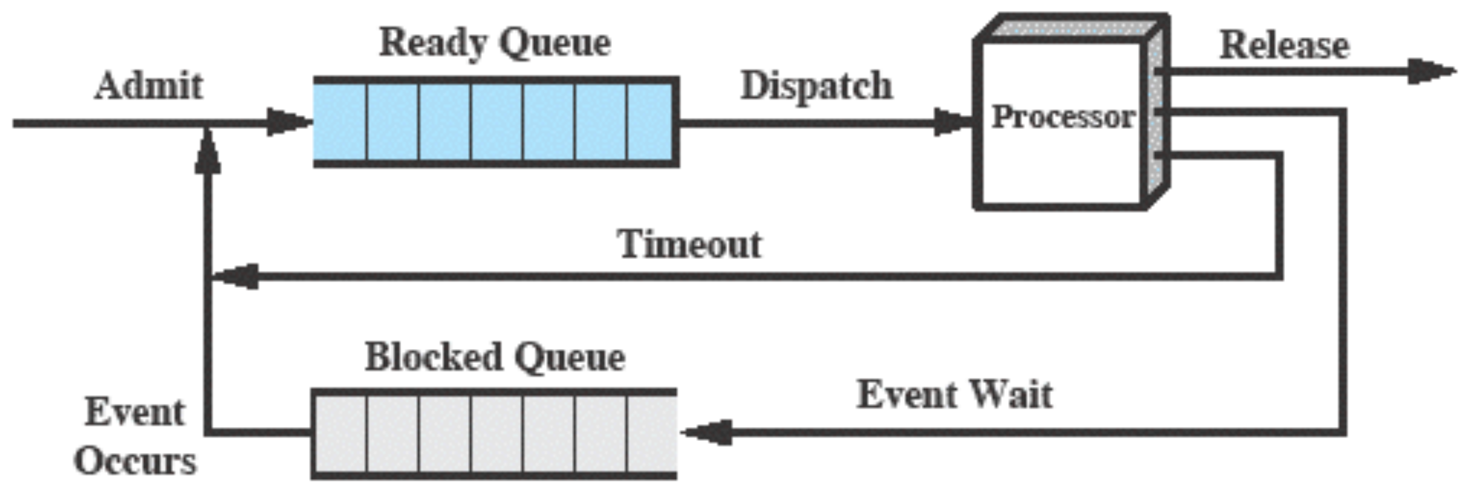


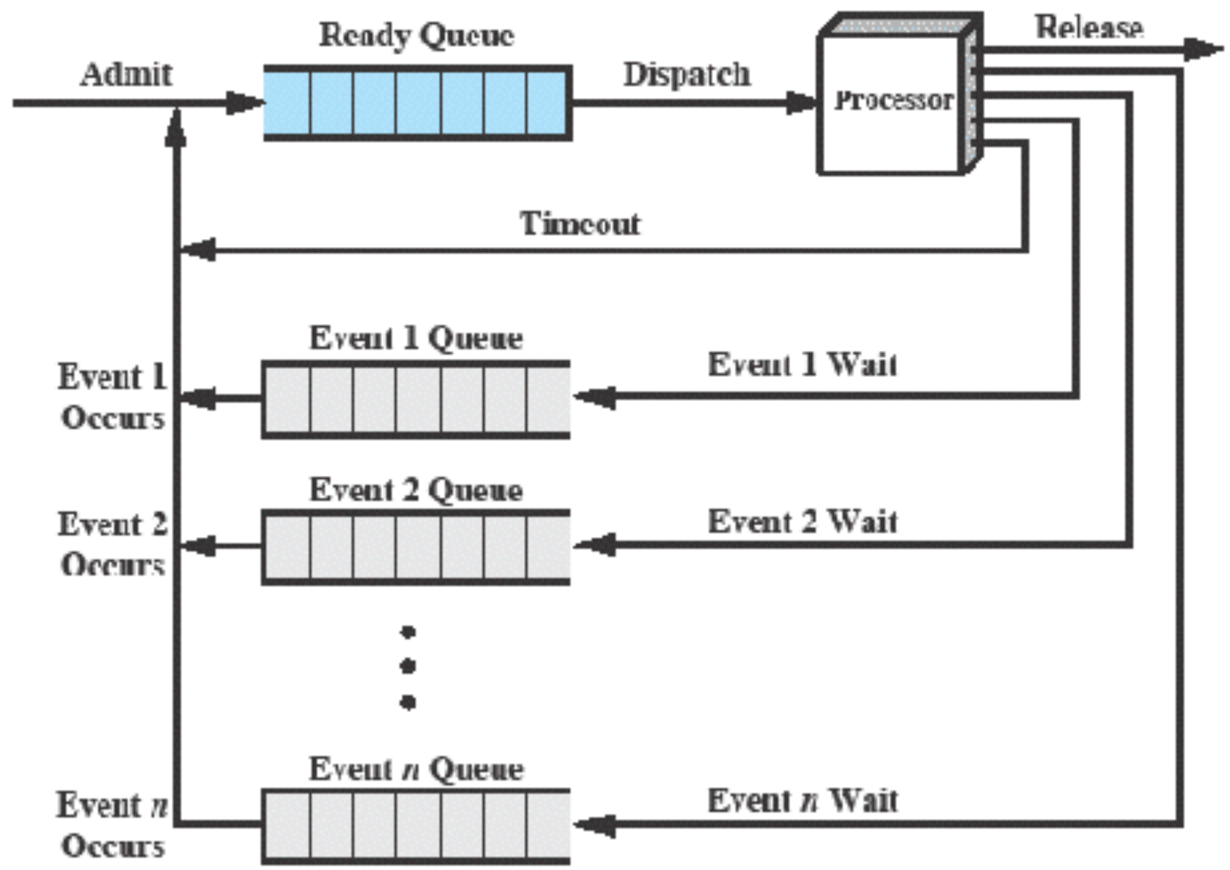
Figure 3.6 Five-State Process Model

Using Two Queues



(a) Single blocked queue

Multiple Blocked Queues

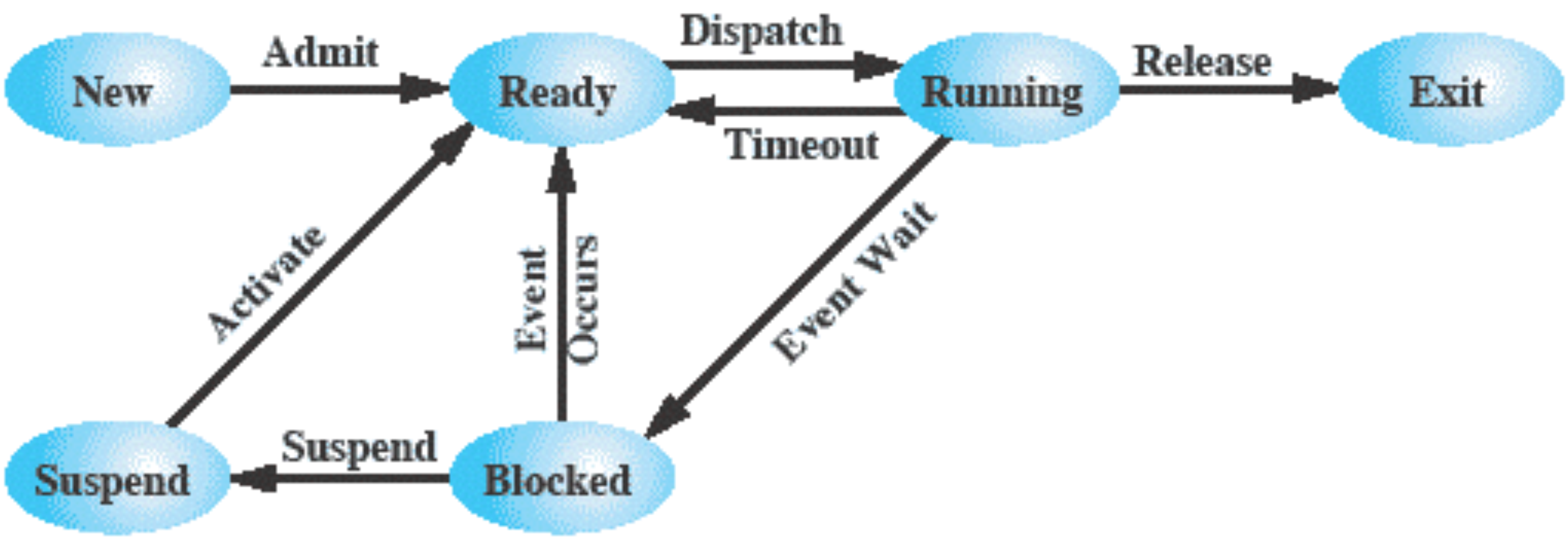


(b) Multiple blocked queues

Five-state process Model

- Five state model works for most discussion
  - Does deadlock if all processes are resident in memory and all waiting for some event to happen
  - This can happen easily due to processor so much faster than I/O
- Helps to add a new state:
  - Suspend: Blocked processes that have been temporarily kicked out of memory to make room for new processes to come in
  - Lets us swap processes to disk to free up more memory
    - Lets us load more processes
  - Blocked state becomes Suspend on getting moved to disk

One Suspend State

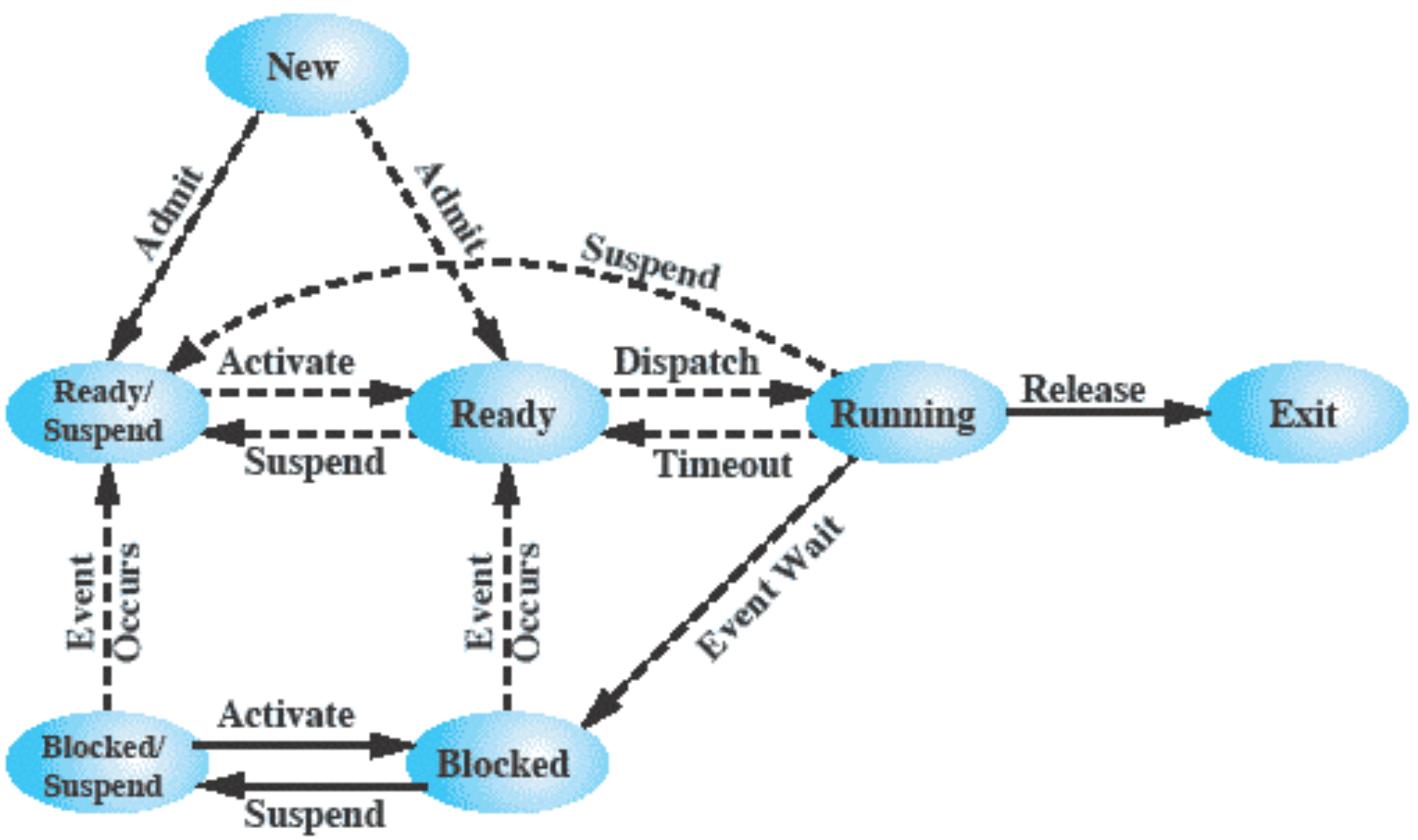


(a) With One Suspend State

Two Suspend states

- Useful to split suspend state up
- Which process to grant CPU when current process is swapped out?
  - Want preference for a previously suspended process over new process (avoid increasing total load)
  - Don't want to ready a process that will be blocked when loaded
- Need to differentiate between:
  - Blocked/Suspend : Blocked on event, put into secondary memory and suspended
  - Ready/Suspend: Ready to run if loaded into memory but suspended

Two Suspend States



(b) With Two Suspend States

Process sleep state

- A process can put itself to sleep while waiting on event
  - Rather than constantly polling for input from keyboard
    - A shell puts itself to sleep
  - Process sleeps in a particular wait channel (WCHAN)
  - When event associated with WCHAN occurs, every process waiting on it is woken up
  - Processes check to see if signal was meant for them
    - For example, set or processes waiting for data from disk
  - If signal is not for them, they put themselves to sleep again

Thundering Herd

- Imagine large number of processes waiting for an event
- Event happens, all processes woken up in response
- Now a race to lock the resource to themselves
  - Remaining ones get put to sleep
- Want to avoid this problem by waking up only one process

Processes and Resources

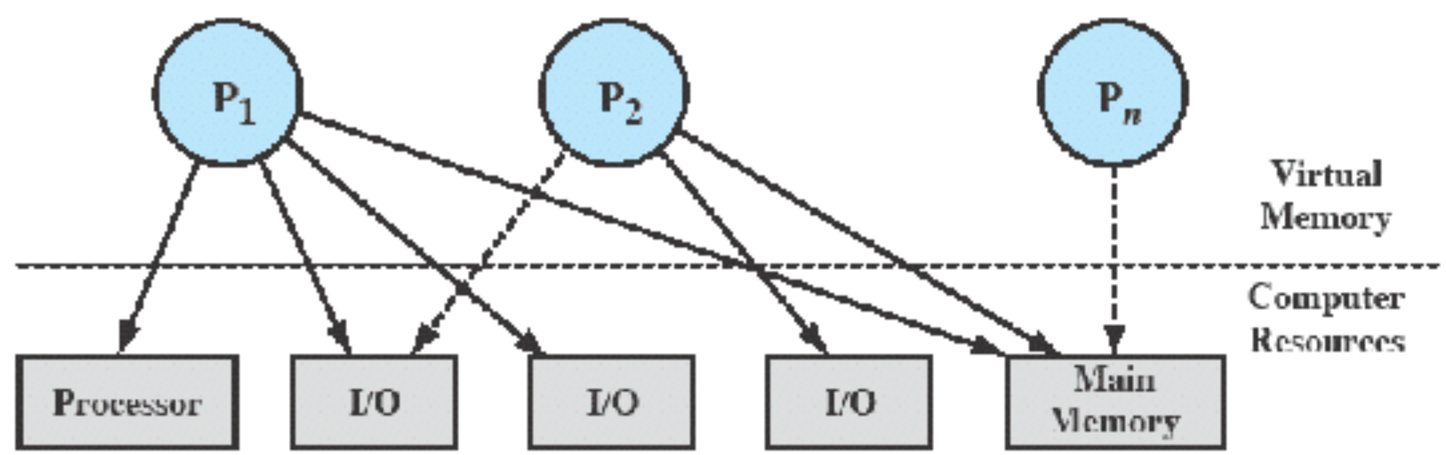


Figure 3.10 Processes and Resources (resource allocation at one snapshot in time)

OS Control Tables

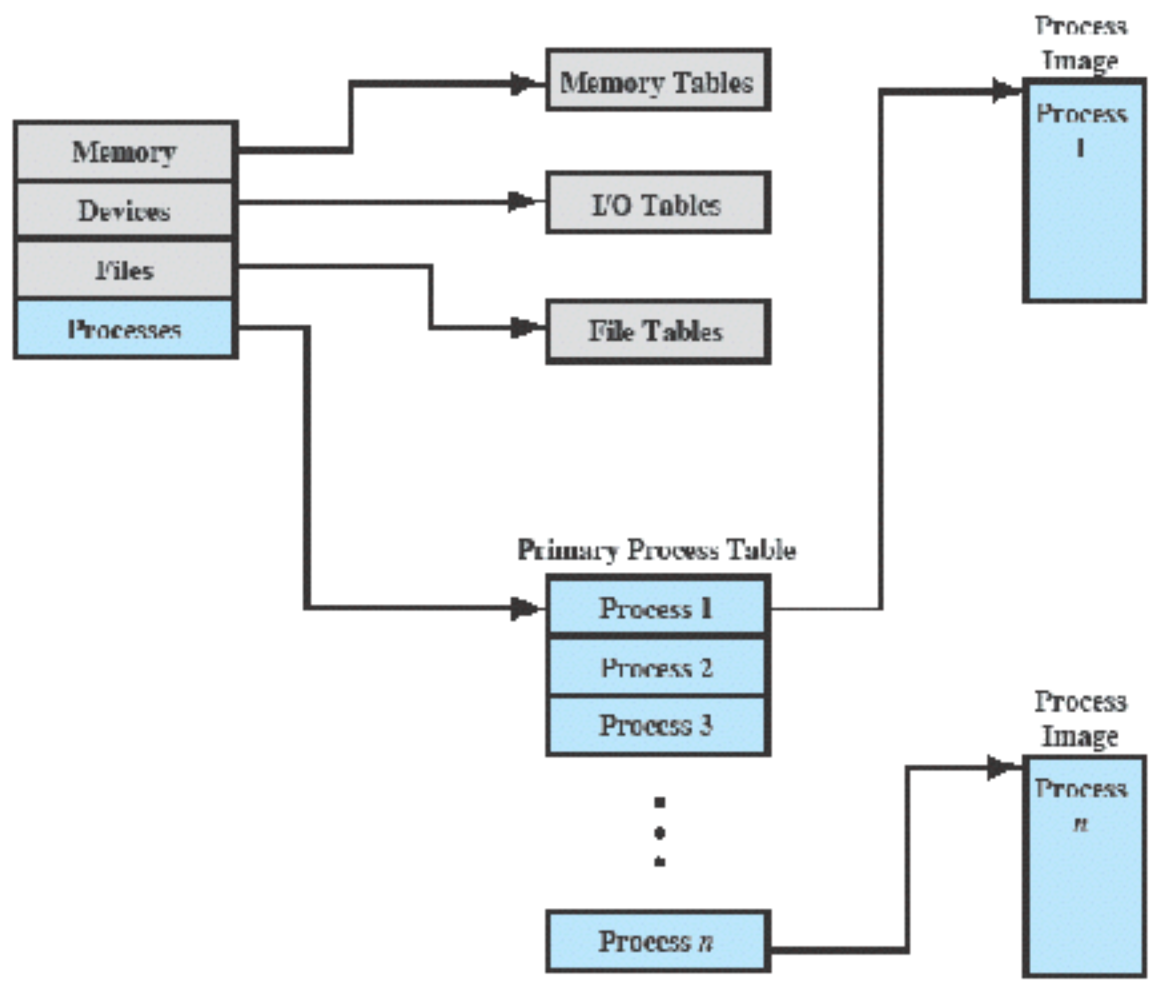


Figure 3.11 General Structure of Operating System Control Tables

Modes of Execution

- Two modes of execution for processes
  - OS execution vs user process execution (kernel mode)
  - OS (kernel) mode allows privileged instructions to be executed
    - Read/write a control register, such as PSW (program status word)
    - Prevents interferences by user code, also doesn't really hurt user programs as they don't usually need this functionality
  - Usually set by bit in the PSW
  - A user process can request to enter kernel mode (then system can set that bit) and then return to user mode

Process Implementation

- Process table
  - One entry for each process
  - Stack pointer
  - Memory allocation
  - Open files
  - Accounting and scheduling information
- Interrupt vector
  - Contains address of *interrupt service procedure*
    - Saves all registers in the process table entry
    - Services the interrupt

Process Creation

- Assign a unique PID to process and add this to the system process table that contains one entry for each process
- Allocate space for all elements of process image
  - Space for code, data, user stack (Could be set by default or based on parameters when created)
- Allocate resources like CPU time, memory, files with some policy:
  - New process obtains resources directly from the OS
  - or new process constrained to share resources from a subset of parent process

Process Creation

- Build the data structures that are needed to manage the process
  - Especially PCB (Process Control Block)
- Initialization data (input)
- Process execution
  - Does parent continue to execute concurrently with children?
  - Parent could wait until all its children have terminated

User processes in memory

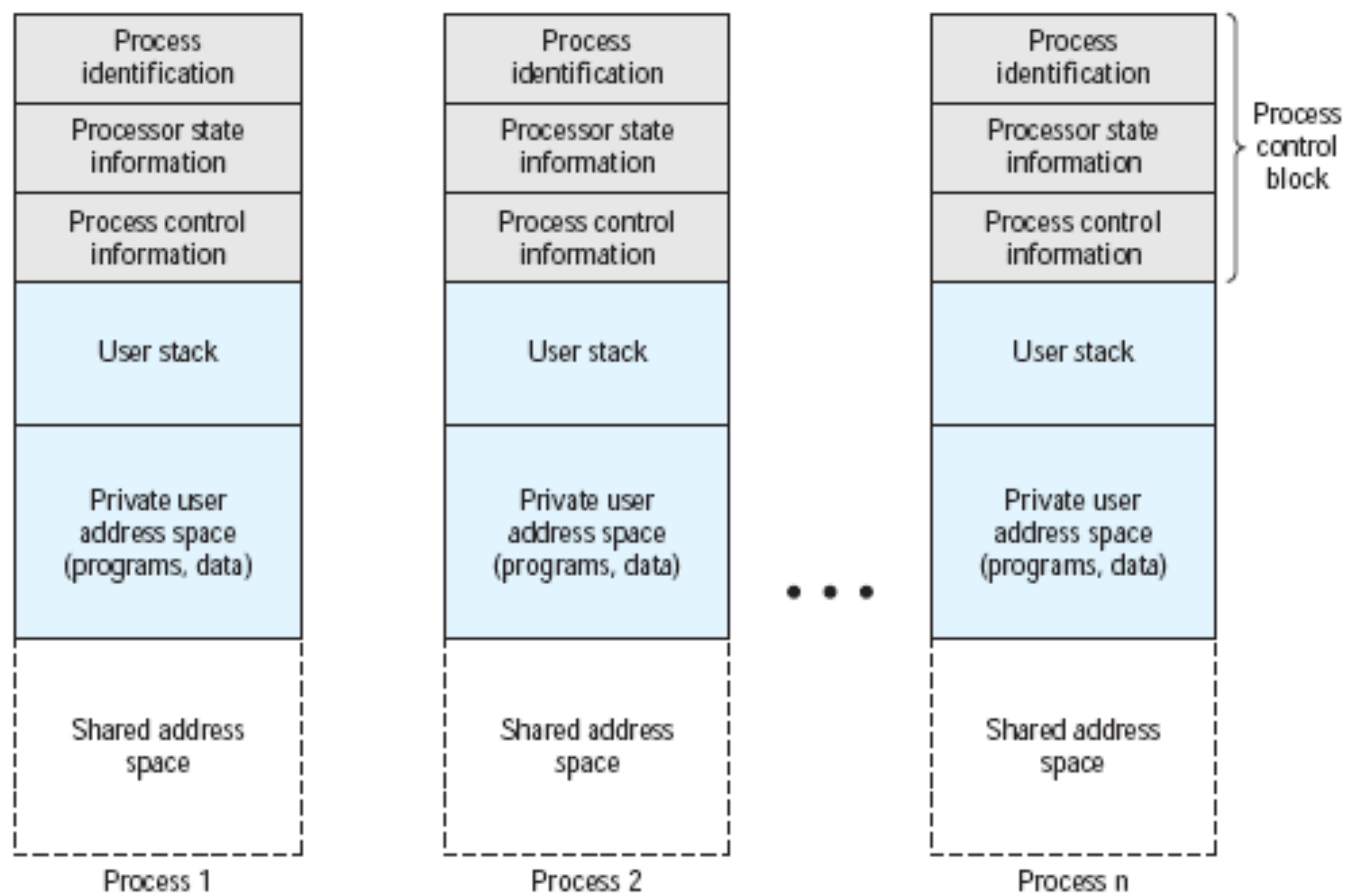


Figure 3.13 **User Processes in Virtual Memory**

Process switching

- Interrupt a running process and assign control to different process
- Difference between process switching and mode switching
  - Mode switching is when going from user to kernel for a bit
  - Process switching is when a new process goes to running
- When do we switch?
  - Any time when the OS has control of the system
  - OS can acquire control by:
    - Interrupt: asynchronous external event, not dependent on instructions, such as clock interrupt
    - Trap - Exception handling; associated with instruction execution
    - Supervisor call - Explicit call to OS

Process switching

Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function

- Mechanisms for interrupting the execution of a process

Change of process state

- Steps in a process switch:
  - Save context of processor including program counter and other registers
  - Update the process control block of the process that is currently in the Running state
  - Move process control block to appropriate queue
    - Ready; Blocked; Ready/Suspend

Change of process state

- Steps in a process switch:
    - Select another process for execution
    - Update the process control block of the process selected
    - Update memory-management data structures
    - Restore context of the selected process

Mode switch

- Anytime that an interrupt happens, could also be a mode switch
- Steps in a mode switch:
  - Save context of processor including program counter and other registers
  - Do any necessary work
  - Restore state of CPU and change PC so old process resumes
- Takes less time than a process switch
- Could switch back to previous process without changing its state
  - Without taking it out of running state

Processes in Unix

- Identified by unique integer - process identifier (PID)
- Created by the `fork(2)` system call
    - `man 2 fork` or just `man fork` (Number is section of man page it is assigned to)
    - Copies 3 segments (instructions, user-data, system data) without initialization from program
    - New process is copy of address space of original process
        - Allows easy communication of parent process with its child
    - Both processes continue execution at instruction after fork
    - Return code for fork is zero for child process, PID of the child for parent process

Fork in Unix

- OS, in kernel mode, in a bit more detail:
  - Allocates slot in process table for new process
  - Assigns unique process ID to child
  - Copy’s process image of parent, with exception of shared memory
  - Increment counters of any file owned by parent
    - Reflects additional process now also owns these files
  - Assign child process to the Ready or Run state
  - Returns ID of child to parent, value of 0 to child process

Fork in Unix

- Both parents data and code need to be duplicated in the copies assigned to child
- Not very efficient to make copies
  - Most of the time, fork may be followed by exec call
- Can save on this by hardware paging
  - Kernels use Copy-On-Write approach to defer page duplication until last possible moment
    - When parent or child need to write into the page

Processes in Unix

- `Exec (2)` system call
  - Usually used after fork
  - Replaces child processes memory space with new program
    - Overlay image of a program onto running process
    - Reinitialize a process from a designated program
    - Program changes but the process remains

Processes in Unix

- `exit(2)` system call
  - Finish executing a process
  - Kernel releases resources owned by the process
  - All open stdio streams are flushed and closed
  - Sends a SIGCHLD signal to parent

Processes in Unix

- `wait (2)` system call
  - Wait for child process to stop or terminate
  - Used to synchronize process execution with `exit` of a previously `fork`ed process
- `brk (2)` system call
  - Change amount of space allocated for the calling processes data segment
  - Changes location of program break, which defines end of process's data segment
  - Control the size of memory allocated to a process

Processes in Unix

- `signal (3)` library function
  - Control process response to extraordinary events
  - Complete family of `signal` functions available
    - sigaction, sigsuspend, waitid, etc

Daemons or kernel threads

- Background processes to do useful work on behalf of user
    - Sit in machine, doing its task
- Some are privileged processes in Unix
    - Run in kernel mode in kernel address space
- Differ in having no `stdin` or `stdout`, sleep most of the time
    - Communicate with humans through log files

Daemons or kernel threads

- Created during system startup and remain alive until the system is shut down
- Common daemons are:
  - `update` to synchronize file system with image in kernel memory
  - `cron` for general purpose task scheduling
  - `lpd` or `lpsched` as a line printer daemon to pick up files scheduled for printing and distributing them to printers
  - `init` is the first started process, is ancestor of all processes
    - Starts one process after another (I/O blocking issues)
  - `swapper` handles kernel requests to swap pages of memory to/from disk

Processes in Unix

- Orphan process
  - Process whose parent process has finished or terminated and is still running itself
  - Adopted by `init` process usually
- Zombie process
  - Processes waiting to send a message to parent so that they can die
  - Completed execution, still has entry in process table
    - End result could be a resource leak from the process table
  - `init` routinely issues `wait (2)` system call whose side effect is to get rid of all orphaned zombies

Processes in Unix

- Wait queues
  - Represent sleeping processes to be woken up by kernel when a condition becomes true
  - Used for interrupt handling, process synchronization and timing
  - Disk operation to terminate, system resource to be released, fixed interval of time to elapse
  - A process waiting for a specific event is put into corresponding wait queue
  - Modified by interrupt handlers and major kernel functions
    - Must be protected from concurrent access
    - Synchronization achieved by spin lock in the wait queue head

Process termination in Unix

- Normal termination
  - Process terminates when it executes last statement
  - Upon termination, the OS deletes the process
  - Process may return data (output) to its parent
- Abnormal termination
  - Process terminates by executing library function `abort (3c)`
  - All file streams are closed and other housekeeping performed as defined by signal handler

Process termination in Unix

- Termination by another process
    - Termination by the system call kill with the signal SIGKILL
    - Usually terminated only by parent of the process because:
        - Child may exceed the usage of its allocated resources
        - Task assigned to the child is no longer required
- Cascading termination
    - Some systems do not allow child to exist if parent has terminated
    - Initiated by the OS

Process termination in Unix

- Process removal (background)
  - A process can query the kernel to get execution state of children
  - A process can create a child process to perform a specific task and `wait` to check whether the child has terminated
  - Termination code of child tells parent process whether task was completed

Process termination in Unix

- Process removal
  - Due to previous design choices, Unix kernel is not allowed to discard data in a PCB right after the process terminates
  - Has to wait till parent issues `wait` that refers to terminated process
  - `EXIT_ZOMBIE` state: Process is technically dead but its descriptor must be saved until parent has received notification
  - If parent is dead, the orphan becomes a child of `init` who destroys zombies by issuing a `wait`

Process states in Linux

- Described by six flags and are mutually exclusive
- TASK_RUNNING
  - Either is running or ready to run
- TASK_INTERRUPTIBLE
  - Process is waiting for an event or signal from another process
    - Possible to wake it up
  - Changes to TASK_RUNNING when that happens

Process states in Linux

- TASK_UNINTERRUPTIBLE
  - Still in a wait state like TASK_INTERRUPTIBLE
  - But delivering a signal to this sleeping process leaves it state unchanged
  - Must be explicitly woken up by a specific event
    - Device driver can’t be interrupted until done probing for hardware device
  - For example, cannot be killed by sigkill
  - What if killing a process might end up with corrupted kernel?

Process states in Linux

- TASK_STOPPED
  - Process execution is stopped
  - Result of receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU
- TASK_TRACED
  - Process stopped by a debugger

Process states in Linux

- EXIT_ZOMBIE
  - Process finished execution but parent has not yet issued a `wait` system call
- EXIT_DEAD
  - Process being removed after the parent has just issued a `wait` system call
  - Changing state from `EXIT_ZOMBIE` to `EXIT_DEAD` avoids race conditions due to other threads of execution that execute wait- like calls on the same process

Processes in MS-DOS

- Created by system call to load a specified binary file
    - Loads it into memory and executes it
- Parent is suspended and waits for child to finish execution

Process Control Subsystem in Unix

- Significant part of the Unix kernel
  - Along with file system
- Contains three modules
  - Interprocess communication
  - Scheduler
  - Memory management

Concurrency

- Management of processes and threads is central to OS design
    - Multiprogramming
        - Management of multiple processes within a uniprocessor system
    - Multitasking
        - Management of multiple processes by interleaving their execution on a uniprocessor system, possibly by scheduling
    - Multiprocessing
        - Management of multiple processes within a multiprocessor
    - Distributed processing
        - Multiple processes executing on multiple distributed systems (Clustering)

Concurrency

- Concurrency encompasses host of design issues
  - Communication among processes
  - Sharing and competing for resources
  - Synchronization of activities of multiple processes
  - Allocation of CPU time to processes
- Concurrency arises with
  - Multiple applications - Need to share processing time
  - Structured applications - Single application with multiple modules
  - OS structure - OS implemented as set of processes or threads

Concurrency

- So how do we use concurrency in our own programs?
- One paradigm is `cobegin/coend` paradigm
- Explicitly specify a set of program segments that could be executed concurrently

```
COBEGIN
   p_1;
   p_2;
   ...
   p_3;
COEND;
```

```
cobegin
   t_1 = a + b;
   t_2 = c + d;
   t_3 = e / f;
coend
t_4 = t_1 * t_2;
t+5 = t_4 - t_3;
```

- No major language has this built in by default
  - Some experimental ones do

Concurrency

- `fork, join,` and `quit` primitives
  - More general than `cobegin/coend`
  - `fork x`
    - Creates a new process `q` when executed by process `p`
    - Starts execution of process `q` at instruction labeled `x`
    - Process `p` executes at instruction following the fork
  - quit
    - Terminates process that executes this command

Concurrency

- `join t, y`
  - Provides an indivisible instruction
  - Provides the equivalent of test-and-set instruction in concurrent language
    - Instruction used to write a memory location and return its old value as a single atomic instruction

```
// if old lock is 0, this function ends
// if lock is 1, spin
function Lock(boolean *lock) {
   while (test_and_set(lock) == 1)
   ;
}
```

Concurrency

- Lets look again at our previous example

```
   m = 3;
   fork p2;
   fork p3;
p1 : t1 = a + b; join m, p4; quit;
p2 : t2 = c + d; join m, p4; quit;
p3 : t3 = e / f; join m, p4; quit;
p4 : t4 = t1 * t2;
   t5 = t4 - t3;
```

Concurrency

- In parallel programming language Threading Building Blocks (TBB)
  - Consider a serial loop:

```
for (int i = 0; i < 10000; i++)
   a[i] = f(i) + g(i);
```

  - Parallel loop in Intel's TBB:

```
tbb::parallel_for ( 0, 10000, [&](int i)
   { a[i] = f(i) + g(i); }
);
```

  - `parallel_for` creates task that apply loop body to elements in range

Interprocess Communication

- Race condition
  - A race condition occurs when two processes (or threads) access the same variable/resource without doing any synchronization
    - One process is doing coordinated update of several variables
    - Second process observing one or more of those variables will see inconsistent results
  - Final outcome dependent on the precise timing of two processes

Interprocess Communication

- Example of a race condition:
    - One process is changing the balance in a bank account
        - Another is simultaneously observing the account balance and last activity data
        - Consider what happens when process changing the balance gets interrupted after updating the last activity date but before updating balance
        - If the process reads the data at this point, it does not get accurate information (either in the current or past time)

Interprocess Communication

- OS concerns from concurrency
  - Keeping track of different processes through PCBs
  - Allocating and deallocating various resources for active processes
    - CPU time, memory, files, I/O devices
  - Protecting data and physical resources of each process against unintended or deliberate interference by other processes
  - Functioning of a process and its I/O which proceed at different speeds, relative to the speed of other concurrent processes

Process interaction

Table 5.2 Process Interaction

Degree of Awareness	Relationship	Influence That One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

Interprocess Communication

- Three main problems in concurrency
  - Need for mutual exclusion
    - **Critical sections**
  - Deadlock
  - Starvation

Critical Section Problem

- Section of code that modifies some memory/file/table while assuming its exclusive control
- Mutually exclusive execution in time
- template for each process that involves critical section

```
do {
...                         /* entry section;         */
critical_section();         /* Assumed to be present */
...                         /* Exit section           */
remainder_section();        /* Assumed to be present */
} while ( 1 );
```

- We have to fill in the gaps of ... for entry and exit sections in this template and test the resulting program for compliance with our protocol specified next

Critical Section Problem

- Design of protocol to be used by processes should cooperate with following constraints:
  - Mutual Exclusion - If process p_i is executing in critical section, then no other processes can be executing in their critical sections
  - Progress - If no process is executing in its critical section, the selection of a process that will be allowed to enter its critical section cannot be postponed
  - Bounded waiting - There must exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Critical Section Problem

- Assumptions
  - No assumption about hardware instructions
  - No assumption about the number of processors supported
  - Basic machine language instructions are executed atomically

Critical Section Problem

- What about disabling interrupts before going into a critical section?
  - Brute-force approach
  - Not proper to give users the power to disable interrupts
    - User may not enable interrupts after being done
    - What about multiple CPU configuration?
  - In current systems, interrupts must be disabled inside some critical kernel regions
    - Critical regions must be limited as kernel and interrupt handlers should be able to run most of the time to take care of any event

Attempt 1

- What about using a lock variable?
  - Share a variable that is set when a process is in its critical region
  - Initially set to zero
  - Going into critical region, first test the lock
    - If zero, process sets to 1 and continues in critical region
  - Can we have a race condition?

Attempt 1

- What about using a lock variable?
  - Definitely race condition
    - A sees lock is 0 but in the middle of this, B then interrupts and runs and sets lock to 1. Then when A runs again, it sets lock to 1. Both in critical region.

Attempt 2

- Strict alternation
  - Use shared variable turn keeps track of whose turn it is to enter critical region.
    - Initially process A inspects turn, finds it to be 0, enters CR
    - Process B finds it 0, sits in a loop testing turn and waiting for a 1
      - Called *Busy-Waiting*

Attempt 2

- Strict alteration

```
extern int turn;     /* Shared variable between both processes*/
void process( const int me) { /* me can be 0 or 1 */
int other = 1 - me;
do {
   while (turn != me);    /* Busy wait*/

      critical_section();
      turn = other;   /*  Release */
      remainder_section();
   }
}
```

- Does not satisfy progress requirement
- Does not keep sufficient information about state of each process

Attempt 3

- What about the use of a flag vector?

```
extern int flag[2]; /* Shared variable; one for each process */

void process ( const int me ) { /* me can be 0 or 1 */
   int other = 1 - me;
   do{
      flag[me] = 1; /* true */
      while ( flag[other] );
      critical_section();
      flag[me] = 0; /* false */
      remainder_section();
   } while ( 1 );
}
```

- Does it satisfy mutual exclusion?

Attempt 3

- Does it satisfy mutual exclusion?

```
Time T0: p0 finds flag[1] set to false
Time T1: p1 finds flag[0] set to false
Time T0: p0 sets flag[1] to true, goes in
Time T1: p1 sets flag[0] to true, goes in
```

  - Processes p_0 and p_1 loop forever in their while statements

- Critically dependent on the exact timing of two processes

Attempt 3

- Does it satisfy progress requirement?

```
Time T0: p0 sets flag[0] to true
Time T1: p1 sets flag[1] to true
```

  - Processes p_0 and p_1 loop forever in their while statements

- Critically dependent on the exact timing of two processes

Attempt 4

- Issue in previous examples is just not enough information stored
- Process should not set its state without knowing state of other processes
  - A process insists on its right to go, then goes in
  - No chance to back off
- What if we change it so a process indicates it wants to go in
  - But is willing to prepare to reset
- Then would have mutual exclusion, problem in practice though

Attempt 4

- This solution is just too courteous
  - What if they both keep backing off in sequence
    - "You first." "No you first." "No, you first"

```
Time T0: p0 sets flag[0] to true   // p0 wants in
Time T1: p1 sets flag[1] to true   // p1 wants in
Time T2: p0 checks flag[1]         // Oops, p1 wants in
Time T3: p1 checks flag[0]         // Oops, p0 wants in
Time T4: p0 sets flag[0] to false  // p0 tries to let p1 in
Time T5: p1 sets flag[1] to false  // p1 tries to let p0 in
Time T6: p0 sets flag[0] to true   // p0 wants in again
Time T7: p1 sets flag[1] to true   // p1 wants in again
```

Attempt 4

- It is possible to make this solution work
- Impose an ordering
  - Akin to how at a mutual stop, person to right goes first
  - Known as Dekker's Algorithm
- Algorithm is a bit hard to prove that it works though
- Lets look at a solution that does work and is popular

Peterson's solution

- Lets try combining `turn` and `flag`!

```
extern int flag[2]; /* Shared variable; one for each process */
extern int turn;    /* Shared variable */

void process ( const int me ) { /* me can be 0 or 1 */
   int other = 1 - me;
   do{
      /* Entry section */
      flag[me] = 1; /* true */
      turn = other   /* cede the turn */
      while ( flag[other]  && turn == other);

      critical_section();

      flag[me] = 0; /* false */

      remainder_section();

   } while ( 1 );
}
```

Peterson's solution

- Idea is to take turns only if somebody else is interested; otherwise go!
- Does it meet our criteria?
- Mutual Exclusion
  - If `p0` is in CR, then `flag[0]` is true
  - Also, either:
    - `flag[1]` is false (`p1` out of critical region) or
    - turn is 0 (meaning `p1` is waiting to enter critical region) or
    - P1 is trying to enter critical region
  - So only way both processes can be in critical region is if `flag[0]` and `flag[1]` are true and `turn = 0` and `turn = 1`

Peterson’s solution

- Progress
  - A process cannot immediately re-enter critical region if the other process has set a flag waiting to go in
- Bounded Waiting
  - A process will wait no longer than one turn
  - After giving priority to other process, process will run to completion and set its flag to 0, so other can enter critical region
- Only works for 2 processes, can we generalize?
  - Filter algorithm

Multiple process solution

- This time, array `flag` takes one of three values (`idle`, `want-in`, `in-cs`)

```
enum state { idle, want_in, in_cs };
extern int turn;
extern state flag[n];/*Flag corresponding to each process in shared memory */

process(const int i ) {
   do {
      do {
         flag[i] = want_in; // Raise my flag
         j = turn; // Set local variable
         // wait until its my turn
         while ( j != i )
            j = ( flag[j] != idle ) ? turn : ( j + 1 ) % n;

         // Declare intention to enter critical section
         flag[i] = in_cs;

         // Check that no one else is in critical section
         for ( j = 0; j < n; j++ )
            if ( ( j != i ) && ( flag[j] == in_cs ) )
               break;
      } while ( ( j < n ) || ( turn != i && flag[turn] != idle ) );

   // Assign turn to self and enter critical section
   turn = i;
   critical_section();

   // Exit section
   j = (turn + 1) % n;
   while (flag[j] == idle)
      j = (j + 1) % n;

   // Assign turn to next waiting process; change own flag to idle
   turn = j; flag[i] = idle;

   remainder_section();
   } while ( 1 );
}
```

Multiple processor solution

- `pi` enters the critical region only if `flag[j] != in-cs` for all `j != i`
- turn can be modified only upon entry to and exit from critical section
  - First contending process enters its critical section
- Upon exit, successor process is designated to be the one following current process
- Mutual Exclusion?
  - `pi` enters the cr only if `flag[j] != in_cs` for all `j != i`
  - Only `pi` can set `flag[i] = in_cs`
  - `pi` inspects `flag[j]` only while `flag[i] = in_cs`

Multiple processor solution

- Progress
  - turn can be modified only upon entry to and exit from critical section
  - No process is running or exiting critical section so `turn` remains constant
  - First contending process in cyclic ordering (`turn`, `turn+1,..,n-1,0,...,turn-1`) enters its critical section
- Bounded Wait
  - Upon exit from CS, process must designate its unique successor in the ordering
  - Any process waiting to enter CS will do so in at most `n-1` turns

Bakery Algorithm

- Each process has unique id and this id assigned in an ordered manner

```
extern bool choosing[n]; /* Shared Boolean array */
extern int number[n]; /* Shared integer array to hold turn number */

void process_i ( const int i ) /* ith Process */ {
   do
      choosing[i] = true;
      number[i] = 1 + max(number[0], ..., number[n-1]);
      choosing[i] = false;
      for ( int j = 0; j < n; j++ ) {
         while ( choosing[j] ); // Wait if j happens to be choosing

         while ( (number[j] != 0)
           &&  ( number[j] < number[i] || (number[j] == number[i] && j < i) );
      }

      critical_section();

      number[i] = 0;

      remainder_section();
   while ( 1 );
}
```

Bakery Algorithm

- Each process (not in critical region) has a variable that indicates its position in a queue of other processes not in critical region
- Each on trying to get in scans variables of other processes
  - Enters critical section only after determining that it is head of the queue
- When process Pi tries to go into CS, sets its turn to one higher than all others
  - Then it busy-waits until process j is not in middle of choosing a turn
  - Then it busy-waits until Pj's turn is either 0 or greater than Pi
  - Once it goes past all processes like this, it enters CS

Test-and-Set

- Another way is to use synchronization hardware
  - Atomic instruction
  - Disable/enable interrupts to prevent context switches
- Atomic instruction of `test_and_set`
  - Record the old value AND
  - Set the value to indicate availability AND
  - Return the old value

Test-and-Set

- Code of test_and_set

```
int test_and_set (int &target) {
   int tmp;
   tmp = target;
   target = 1;  /* True*/
   return ( tmp );
}
```

- Mutual exclusion with test_and_set

```
extern bool lock ( false );

do
   while ( test_and_set ( lock ) );

   critical_section();

   lock = false;

   remainder_section();
while ( 1 );
```

Producer/Consumer problem

- General situation
  - One or more producers are generating data and placing it in buffers
  - A single consumer is taking items out of buffer one at a time
  - Only one producer or consumer may access buffer at any one time
- The problem:
  - Ensure that the producer can’t add data into full buffer and consumer can’t remove data from empty buffer

Producer/Consumer problem

- Functions that could be used for this problem:
  - Assume circular finite buffer `b` with linear array of elements

Producer	**Consumer**

while (true) {
    /* produce item v */
    while ((in + 1) % n == out)
    /* do nothing */;
    b[in] = v;
    in = (in + 1) % n
}
	while (true) {
	        while (in == out)
	                /* do nothing */;
	        w = b[out];
	        out = (out + 1) % n;
	        /* consume item w */
	}

Semaphores

- Want to solve this problem
- We will do this using semaphores
  - Integer variable that can only be accessed through two standard atomic operations
    - wait(P) and signal(V)

Operation	Semaphore	Dutch	Meaning
Wait	P	proberen	test
Signal	V	verhogen	increment

Semaphores

- Classical definitions for `wait` and `signal` are:

```
wait ( S ):   while (S <= 0);
              S--;


signal ( S ): S++;
```

- Mutual exclusion implementation with semaphores

```
do
   wait( mutex );

   critical_section();

   signal( mutex );

   remainder_section();
while ( 1 );
```

Semaphores

- Also simple matter to enforce synchronization with processes
- Suppose want p_1 to complete a task and then p_2 to start:

p_1	`S`$_1$`;` `signal( synch);`
p_2	`wait( synch );` `S`$_2$`;`

Semaphores

- So how do we actually implement semaphore operations?
- Can use binary semaphores using `test_and_set`
  - As per previous definition
- Implementation with a *busy-wait*

Binary Semaphore

```
class bin_semaphore {
   private:
      bool s;    /* Binary semaphore */

   public:
      bin_semaphore()    // default constructor
      : s ( false ) {}

      void P () {        // sem_wait
         while (test_and_set( s ) );
      }

      void V () {        // sem_signal
         s = false;
      }
};
```

General semaphore

```
class semaphore {
   private:
      bin_semaphore mutex;
      bin_semaphore delay;
      int count;

   public:
      void semaphore ( const int num = 1 )  // Constructor
      : count ( num ) {
         delay.P();   //wait on the delay binary semaphore
      }
      // SemWait
      void P() {
         mutex.P();    // wait on mutex bin sem
         if ( --count < 0) {
            mutex.V();   // signal on mutex bin sem
            delay.P();   // wait on delay bin sem
         }
         else
            mutex.V();    // signal on mutex bin sem
      }
      // SemSignal
      void V() {
         mutex.P();    // wait on mutex bin sem
         if (++count <= 0)
            delay.V();   // signal on delay bin sem
         mutex.V();   // signal on mutex bin sem
      }
};
```

Semaphores

- Both of these are busy/waits
  - Processes waste CPU cycles while waiting to enter critical sections
  - So lets modify `wait` operation into `block` operation
    - Let process block itself rather than busy-wait
  - Modify `signal` operation into `wakeup` operation
  - Change the state of process from wait to ready

BLock-Wakeup protocol

```
// semaphore with block/wait
class sem_int {
   private:
      int value;   // Number of resources
      queue<pid_t>  list;    // List of processes

   public:
      void sem_int ( const int n = 1 )  // Constructor
      : value ( n ) {
         list = queue<pid_t>( 0 );
      }
      // Sem_wait
      void P() {
         if ( --value < 0) {
            pid_t p = getpid();
            list.enqueue( p );   // enqueue invoking process
            block(p);
         }
      }      }
      // Sem_signal
      void V() {
         if (++value <= 0) {
            process p = list.dequeue();
            wakeup( p );
         }
      }
};
```

Producer-Consumer solution

```
extern semaphore mutex;           // To get exclusive access to buffers
extern semaphore empty ( n );     // Number of available buffer space
extern semaphore full( 0 );       // Initialized to 0

void producer() {
   do {
      produce ( item );
      empty.P();             // wait on empty
      mutex.P();             // wait on mutex
      put( item );
      mutex.V();             // signal on mutex
      full.V();              // signal on full
   } while (1);
}

void consumer() {
   do {
      full.P();              // wait on full
      mutex.P();             // wait on mutex
      remove ( item );
      mutex.V();             // signal on mutex
      empty.V();             // signal on empty
      consume( item );
   } while ( 1 );
}
```

Thundering Herd

- Imagine large number of processes waiting for an event
- Event happens, all processes woken up in response
- Now a race to lock the resource to themselves
  - Remaining ones get put to sleep
- Want to avoid this problem by waking up only one process

Event Counters

- Solve the producer-consumer problem without requiring mutual exclusion
- Uses special kind of variable with three operations
  - `E.read()` : Return the current value of `E`
  - `E.advance()`:Atomically increment `E` by 1
  - `E.await(v)`: Wait until `E` has a value of `v` or more
- Event counters always start at 0 and always increase

Producer/Consumer EC

- Can solve Producer/Consumer with event counters
- Both produce a sequence number locally
- For producer:
  - Serial number of each thing it has produced
- For consumer:
  - Serial number of next item it will consume

Event Counters

```
class event_counter {
   int ec;    // Event counter

   public:
      event_counter():ec( 0 ) {}    // Default constructor
      int read()  const   {  return ( ec ); }
      void advance()      {  ec++; }
      void await( const int v ) const  {while (ec < v); }
};


extern event_counter   in, out;    Shared event counters
void producer() {
   int sequence (0);          // Local to producer
   do {
      produce( item );
      sequence++;
      out.await( sequence - num_buffers);
      put( item );
      in.advance();
   }
   while ( 1 );
}
```

Event Counters

```
extern event_counter in,out;

void consumer() {
   int sequence ( 0 );    // Local to consumer
   do {
      sequence++;
      in.await( sequence );
      remove( item);
      out.advance();
      consume(item);
   } while ( 1 );
}
```

Semaphores

- Semaphores can be used to solve any traditional synchronization problem
- However, several drawbacks
  - Essentially shared global variables, can be accessed anywhere
  - No connection between semaphore and data being controlled by sem
  - Used both for critical sections (mutual exclusion) and coordination (scheduling)
  - No control or guarantee of proper usage
- Can be hard to use and prone to bugs
  - Really want programming language support

Monitors

- Semaphores are low-lvl
- A monitor is an attempt to create a high-level synchronization primitive
  - Easier to use than semaphores
- A programming construct
  - Part of a programming language and not the OS
  - Implemented by compiler
    - May implement its parts using semaphores

Monitors

- Implementation easiest to view as a class with private and public functions
- Collection of data [resources] and private functions to manipulate data
- A monitor must guarantee the following:
  - Access to resource possible only via one of the monitor procedures
    - Data is essentially local to that procedure
  - Process enters the monitor by invoking one of its public procedures
  - Procedures are mutually exclusive in time
    - Only one process can be active within a monitor at any given time
      - A monitor has a wait queue, since if a process tries to call monitor and it is already used, it is blocked

Monitors

- Monitors let you do very convenient things
  - Lets say you have many linked lists of data
    - You could lock the entire set of linked lists with one monitor (lock)
    - Lock each linked list with a separate lock
    - Lock each element of each linked lists with a separate lock

Monitors

- Very easy to create mutual exclusion around specific things
- Suppose we want to create a counter and not worry about race conditions

```
monitor sharedcounter {
   int counter;
   function add() { counter++;}
   function sub() { counter--;}
   init() { counter=0; }
}
```

Monitors

- While this easy access to exclusion is nice
  - Not only thing that we might want
- For example, in producer/consumer problem
  - Producer wants to signal consumer that buffer is no longer empty
- Many times a process might want to signal another based on some condition
  - Add condition variables to monitors

Monitors

- Has mechanism to enable syncing, similar to blocking semaphore
  - the `condition` construct with `wait` and `signal` operations
    - `x.wait()` suspends process until another process invokes `x.signal()`
    - `x.signal()` resumes exactly one suspended process; it has no effect if no process is suspended
- After signal, it must select a process to execute within monitor
  - Suppose `x.signal()` executed by process `P` allowing the suspended process `Q` to execute, can go with two ways
    - `Q` immediately starts executing, if `P` is not done then `Q` gets blocked
      - Extra context switches (Advocated by Hoare)
    - `Q` waits until `P` leaves monitor, or waits for another condition

Dining Philosophers

- 5 philosophers sit at a table with bowls of spaghetti
- Each philosopher must think and then eat, alternating
- Can only eat if they have right and left chopsticks
- How to design it so no philosopher starves

Dining philosophers

- Seems easy!
  - Think until left fork is available; when it is, pick it up
  - Think until right fork is available; when it is, pick it up
  - When both forks are held, eat for a fixed amount of time
    - Then, put right fork down
    - Then, put left fork down
  - Repeat

Dining philosophers

- Can deadlock
    - State in which each philosopher has picked up fork to left
        - Waiting on fork to the right
    - Eternally waits for each to release fork
- Can lead to starvation
    - One more philosopher that is never ever given both forks
    - Everyone else keeps eating, while he starves to death

Solution by monitors

```
enum state_type {thinking, hungry, eating}

class dining_philosophers {
   private:
      state_type state[5];   // State of each philosopher
      condition self[5];     // Condition object for synching

      void test ( int i ) {
         if (( state[ (i + 4 ) % 5] != eating) &&
            ( state[i] == hungry)       &&
            ( state[ ( i + 1) % 5] != eating ) ) {
            state[ i ] = eating;
            // no effect on pickup but important to wake up during putdown
            self[i].signal();
         }
       }

   public:
      void dining_philosophers() {   // Constructor
         for (int i = 0; i < 5; state[i++] = thinking); }
      void pickup( const int i ) {   // i corresponds to phil i
         state[i] = hungry;
         test( i );   // set state to eating only if neighbors not eating
         if (state[i] != eating)  // if unable to eat, wait
            self[i].wait();
      }
      void putdown( const int i)     // i corresponds to phil i
         state[i] = thinking;     // put down chopsticks
         test( ( i+4) % 5);    //  if R/L is hungry and both are not eating
         test( ( i+1) % 5);    // set Rs state to eating and wake it up
      }
}
```

Solution by monitors

- Philosopher `i` must invoke operations `pickup` and `putdown` on instance `dp` of `dining_philosophers` monitor

```
dining_philosophers dp;

dp.pickup(i);   // Philosopher i picks up the chopsticks
...
dp.eat(i);      // Philosopher i eats (for random amount of time)
...
dp.putdown(i);  // Philosopher i puts down the chopsticks
```

- No two neighbors eating simultaneously - no deadlocks
- However, starvation could occur in this example

Implementing monitors

- Execution of procedures must be mutually exclusive
- A wait must block the current process on the corresponding condition
- If no process is running in the monitor and some process is waiting, it must be selected
  - If more than one waiting process, need some criterion for selecting and deploying one
- Implementation using semaphores
  - Semaphore `mutex` corresponding to the monitor initialized to 1
    - Before entry, execute `wait(mutex)`
    - Upon exit, execute `signal(mutex)`
  - Semaphore `next` to suspend processes unable to enter monitor initialized to zero

Implementing monitors

- Integer variable `next_count` to count number of processes waiting to enter monitor

```
mutex.wait();
...
void proc() { ... }   // Body of process
...
if (next_count > 0)
   next.signal();
else
   mutex.signal();
```

- Semaphore `x_sem` for `condition x`, initialized to zero
- Integer variable `x_num_waiting_procs`

Implementation of monitors

```
class condition {
   int        num_waiting_procs;   // Processes waiting on this condition
   semaphore  sem;                 // To synchronize processes
   static int next_count;          // Processes waiting to enter monitor
   static semaphore next;
   static semaphore mutex;

   public:
      condition() : num_waiting_procs (0), sem (0)   // Default const

      void wait() {
         num_waiting_procs++;     // # of procs waiting on this condition
         if (next_count > 0 )     // Someone waiting inside monitor?
            next.signal();        // Yes, wake him up
         else
            mutex.signal();       // No, free mutex so others can enter
         sem.wait();              // Start waiting on condition
         num_waiting_procs--;     // Wait over, decrement variable
      }

      void signal() {
         if (num_waiting_procs <= 0)     // nobody waiting?
            return;
         next_count++;           // # of ready processes inside monitor
         sem.signal();           // Send the signal
         next.wait();            // You wait; let signaled process run
         next_count--;           // One less process in monitor
      }
};
```

Monitors

- Condition variables != semaphores
  - Operations have same names, they have different semantics
  - However, can be used to implement the other
- Access to monitor is controlled by a lock
  - wait() blocks the calling thread and gives up the lock
    - To call wait, the thread has to be in the monitor (hence has lock)
    - Sem_wait just blocks the thread on the queue
  - signal() causes a waiting thread to wake up
    - If no waiting thread, signal is lost
    - Sem_signal increases semaphore count, allowing future entry
      - Even if no thread is waiting
    - Condition variables have no history

Advantages over semaphores

- Of course possible to make mistakes with monitors
  - Could forget to issue signal after done with a resource
  - Other processes hang up waiting for it
- However, all synchronization functions part of monitor
  - Easier to verify synching has been done correctly.
- Once monitor is coded correctly, safe no matter how many people access it
- With semaphores, resource access only correct if each process uses them correctly

Message-Based Synchronization

- Process interaction involves two things
  - Synchronization (mutual exclusion)
  - Communication (information exchange)
- Communication between processes is achieved by:
  - Shared memory (semaphores, monitors, CCRs)
    - CCR: Concurrency and Coordination Runtime
  - Message Systems
    - Desirable as they prevent sharing, so less security concerns and also maybe lack of shared memory due to different physical hardware

Message-Based Synchronization

- Communication by passing messages
  - Processes communicate without any need for shared variables
  - Paradigm of choice for distributed systems, shared memory multiprocessors and uniprocessors
  - Two basic communication primitives
  - send message:
    - send(P, message) : Send a message to process P
  - receive message:
    - receive(Q, message) : Receive a message from process Q
  - Messages passed through a communication link
    - Physical or logical link between two processes, set up automatically

Message-Based Synchronization

- Issues that need to be resolved
  - Sync vs Asynch communication
    - Upon send, does sending process continue or does it `wait` for it to be accepted by receiving process (synch or blocking communication)
    - What happens if receive is issued and no message waiting (block or not block?)
  - Implicit vs Explicit Naming
    - Does sender specify exactly one receiver or transmit to all other processes

```
send (p, message)     //Send a message to process p
send (A, message)      // Send a message eto mailbox A
```

    - Does the receiver accept from one or accept from any

```
receive (p, message)     //Receive a message from process p
receive (id, message)    //Receive a message from any process
                         // who's pid is id
receive (A, message)     // Receive a message from mailbox A
```

Blocking vs Nonblocking

- Blocking send, blocking receive
  - Both are blocked until message is delivered (referred to as rendezvous)
    - Allows tight synchronization between processes
  - Nonblocking send, nonblocking receive:
    - Neither party is required to wait
  - Nonblocking send, blocking receive
    - Sender can continue, receiver is blocked
    - Probably most useful one
- Queues in message passing can be normal queues or priority queues

Message-Based Synchronization

- Producer/Consumer Problem with messages

```
void producer () {
   while (1 ) {
      produce ( data );
      send(consumer, data);
   }
}
```

```
void consumer () {
   while (1 ) {
      receive( producer, data);
      consume( data );
   }
}
```

Message-Based Synchronization

- Mutual exclusion with message passing

```
/* program mutual exclusion */
const int n = 5;  /* total number of processes */
void P(int i) {
   message msg;

   while (true) {
      receive (box, msg);
      /* critical section */
      send (box, msg);
      /* remainder */
   }
}

void main() {
   create mailbox(box);
   send(box, null);
   parbegin(P(1), P(2), ..., P(n));
}
```

Ports and mailboxes

- Can achieve synching of asynch process by embedding a busy-wait loop, with a non-block receive to simulate the effect of implicit naming
  - Inefficient solution
- Indirect communication avoids inefficiency of busy-wait
  - Make queues holding messages between senders and receivers visible to processes through a *mailbox*

Message-Based Synchronization

- Mailboxes
    - Messages are sent to or taken from a mailbox
    - Most general communication facility between *n* senders and *m* receivers
    - Unique identification for each mailbox
    - Process may communicate with another process by a number of different mailboxes
    - Two processes may communicate only if they have a shared mailbox

Message-Based Synchronization

- Communication link
  - A link is established between a pair of processes only if they share mailbox
  - A link may be associated with more than two processes
  - Between any pair of processors, may be different links, each corresponding to one mailbox
  - A link may be unidirectional or bidirectional

Message-Based Synchronization

- Ports
  - In a distributed environment, `receive` referring to same mailbox may reside on different machine
  - Port is a limited form of mailbox associated with only one receiver
  - All messages originating with different processes but addressed to the same port are sent to one central place associated with the receiver

Signal and IPC in Unix/Linux

- POSIX standard defines about 20 signals, two of which are user definable
  - `SIGUSR1, SIGUSR2`
- Process can react to signals in two ways
  - Ignore the signal
  - Asynchronously execute a signal handler
- If process does not specify one of these two, kernel does default action based on number
  - Terminate process
  - Dump core and terminate process
  - Ignore the signal
  - Suspend process
  - Resume the process if it was stopped

Signal and IPC in Unix

- `SIGKILL` and `SIGSTOP` signals cannot be handled directly by the process or ignored
- IPC resources
  - Shared memory, semaphores, and message queues
  - Acquired by `shmget(2), semget(2)` and `msgget(2)`
  - Persistent: Must be explicitly deallocated by creator, current owner or root
  - `msgsnd(2)` and `msgrcv(2)`
  - Shared memory
    - `shmget(2), shmat(2), shmdt(2)`

Oh so much more coming!

- Any questions?