İSTANBUL
KEMERBURGAZ
ÜNİVERSİTESİ

operating system

Spring 2017

Prof.Dr. Hasan Balik

Student Name : Walid .W. Ramadan mansour

Student ID : 163110469

Email : wild.mansour526@gmail.com

Unix SVR4 (OpenSolaris and illumos distributions) Process and Thread Management

OpenSolaris is a discontinued, open source computer operating system based on Solaris created by Sun Microsystems. It was also the name of the project initiated by Sun to build a developer and user community around the software. After the acquisition of Sun Microsystems in 2010, Oracle decided to discontinue open development of the core software, and replaced the OpenSolaris distribution model with the proprietary Solaris Express.

- **illumos** is a free and open-source Unix operating system. It derives from OpenSolaris, which in turn derives from SVR4 UNIX and Berkeley Software Distribution (BSD). illumos comprises a kernel, device drivers, system libraries, and utility software for system administration. This core is now the base for many different open-sourced OpenSolaris distributions,[3] in a similar way in which the Linux kernel is used in different Linux distributions.

OpenSolaris implements multilevel thread support designed to provide considerable flexibility in exploiting processor resources.

Processes

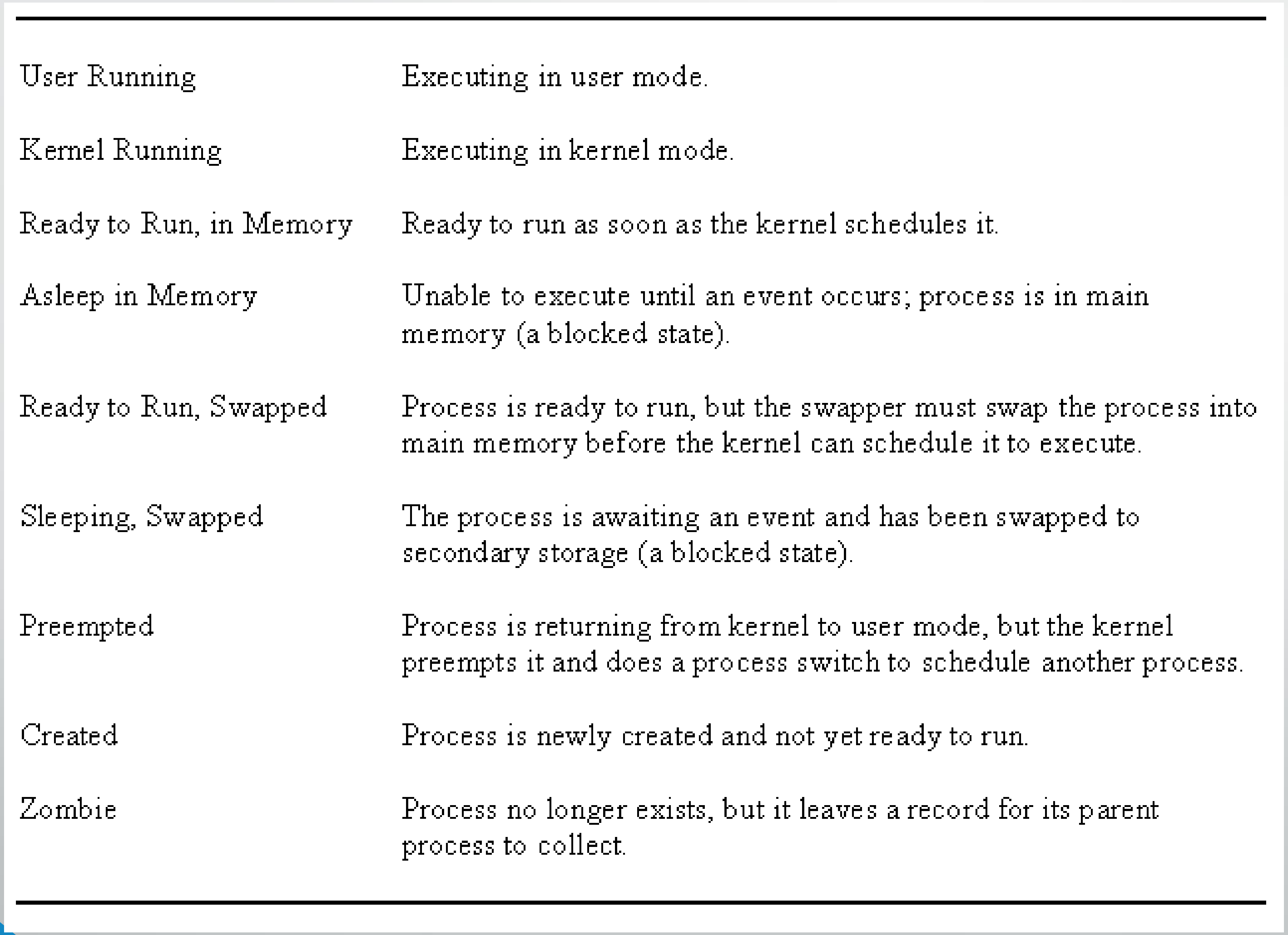
UNIX uses two categories of processes: system processes and user processes. System processes run in kernel mode and execute operating system code to perform administrative and housekeeping functions, such as allocation of memory and process swapping. User processes operate in user mode to execute user programs and utilities and in kernel mode to execute instructions belong to the kernel. A user process enters kernel mode by issuing a system call, when an exception (fault)
is generated or when an interrupt occurs.

Process States

A total of nine process states are recognized by the UNIX operating system
UNIX employs two Running states to indicate whether the process is executing in user

mode or kernel mode. A distinction is made between the two states: (Ready to Run, in Memory) and (Preempted). These are essentially the same state, as indicated by the dotted line joining them. The distinction is made to emphasize the way in which the preempted state is entered. When a process is running in kernel mode (as a result of a supervisor call, clock interrupt, or I/O interrupt), there will come a time when the kernel has completed its work and is ready to return control to the user program. At this point, the kernel may decide to preempt the current process in favor of one that is ready and of higher priority. In that case, the current process moves to the preempted state. However, for purposes of dispatching, those processes in the preempted state and those in the Ready to Run, in Memory state form one queue.

Process States



User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.

Process Description

Process Image

A process in UNIX is a rather complex set of data structures that provide the operating system with all of the information necessary to manage and dispatch processes. summarizes the elements of the process image, which are organized into three parts:

- user-level context.
- register context.
- system-level context.

Unix – User Level Context

Contains the basic elements of a user's program and it is usually generated from a compiled object file

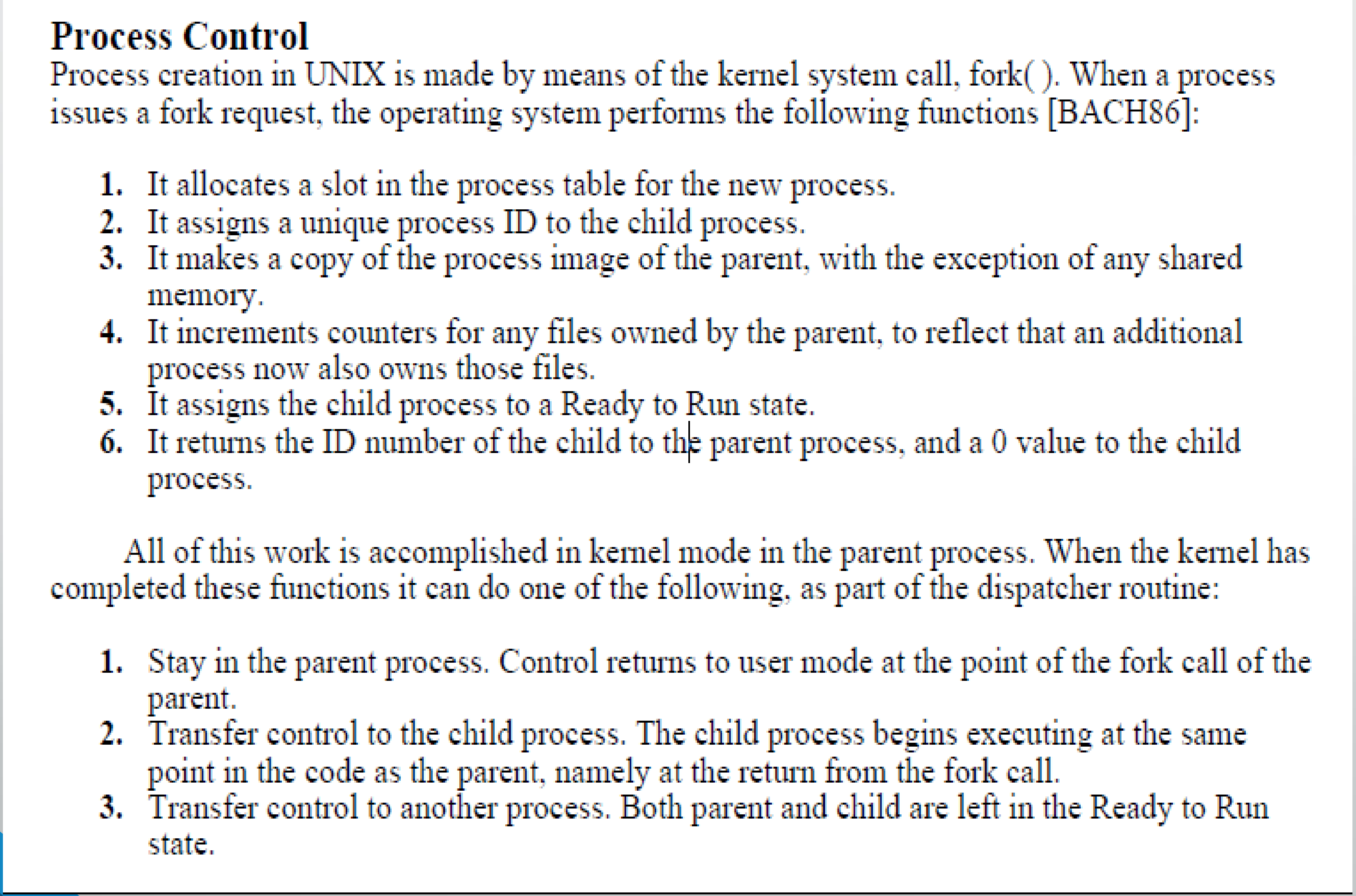
- User Code
  - Read only and is intended to hold the program's instructions
- User Data
  - Data accessible and processed by this process
- User Stack
  - Used while executing in user mode for procedure calls and returns and parameter passing
- Shared Memory
  - Data area shared with other processes; there is just one physical copy of shared area

UNIX – Register Context

- When a process is not running, the processor status information is stored in the register context area
  - Program Counter
    - May be in either user or kernel space
  - PSW (Processor Status Word)
  - Stack Pointer
    - Points to top of user/kernel stack
  - General Registers

UNIX – System Level Context

- Contains the remaining information that the operating system needs to manage the process
- Contains
  - A static part – stays at the same size during a process lifetime
    - Process table entry
    - U area
    - Per process region table
      - Used by the memory management system (contains virtual memory info)
  - A dynamic part
    - Kernel Stack - this stack is used when the process is executing in kernel mode and contains information that must be saved and restored as procedure calls and interrupts occur



Process Control

Process creation in UNIX is made by means of the kernel system call, fork( ). When a process issues a fork request, the operating system performs the following functions [BACH86]:

1. It allocates a slot in the process table for the new process.
2. It assigns a unique process ID to the child process.
3. It makes a copy of the process image of the parent, with the exception of any shared memory.
4. It increments counters for any files owned by the parent, to reflect that an additional process now also owns those files.
5. It assigns the child process to a Ready to Run state.
6. It returns the ID number of the child to the parent process, and a 0 value to the child process.

All of this work is accomplished in kernel mode in the parent process. When the kernel has completed these functions it can do one of the following, as part of the dispatcher routine:

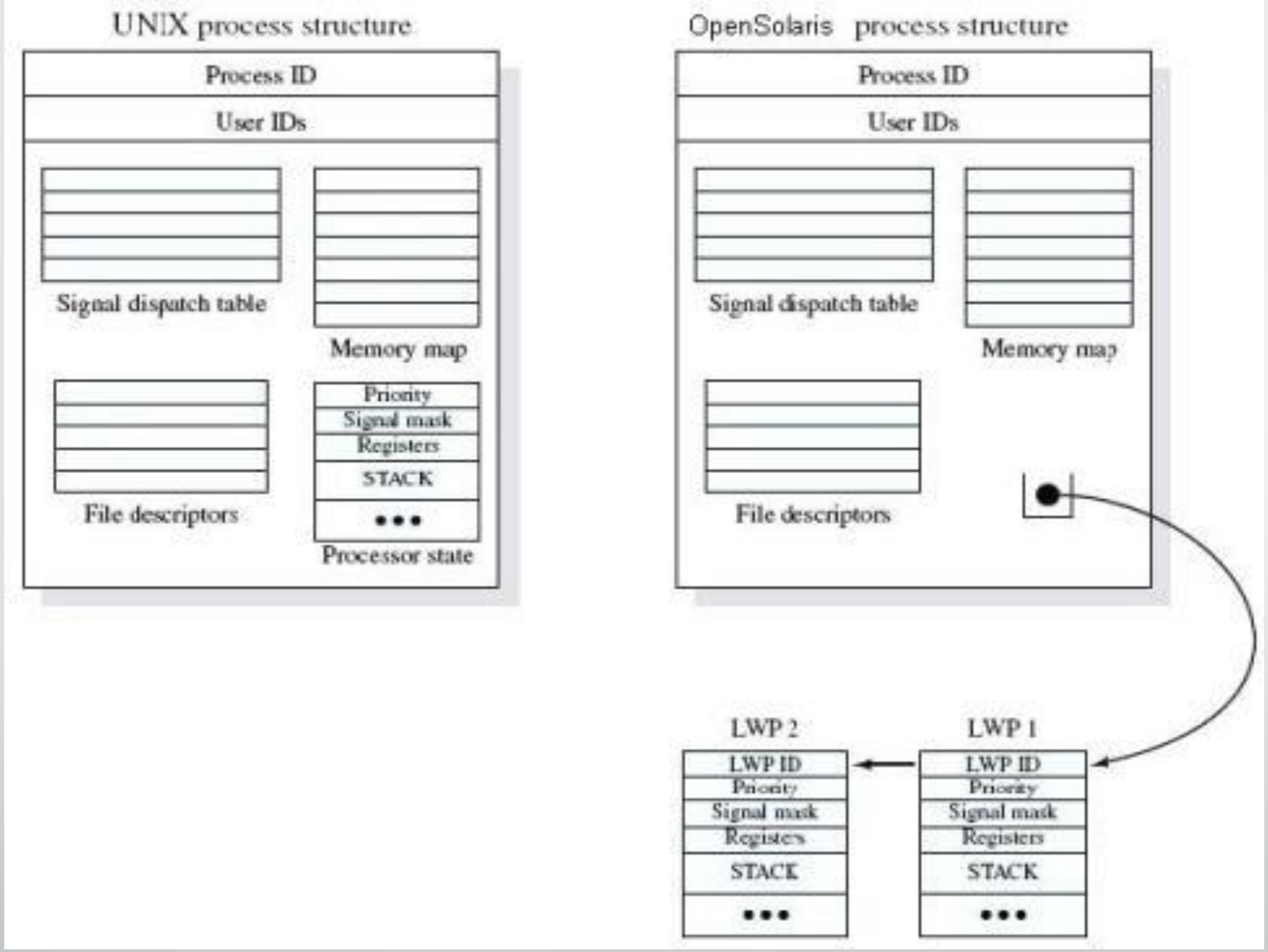
1. Stay in the parent process. Control returns to user mode at the point of the fork call of the parent.
2. Transfer control to the child process. The child process begins executing at the same point in the code as the parent, namely at the return from the fork call.
3. Transfer control to another process. Both parent and child are left in the Ready to Run state.

UNIX - Process Table Entry

- Status – Current state of a process
- Pointers to U area and user code/data
- Process size
- Identifiers (real/effective user/group id)
- Process/Parent ID
- Event Descriptor
- Signal – Signals sent but not handled
- Priority
- Timers - process execution time, user-set alarm
- Memory status – Is it swapped out?

UNIX – U Area

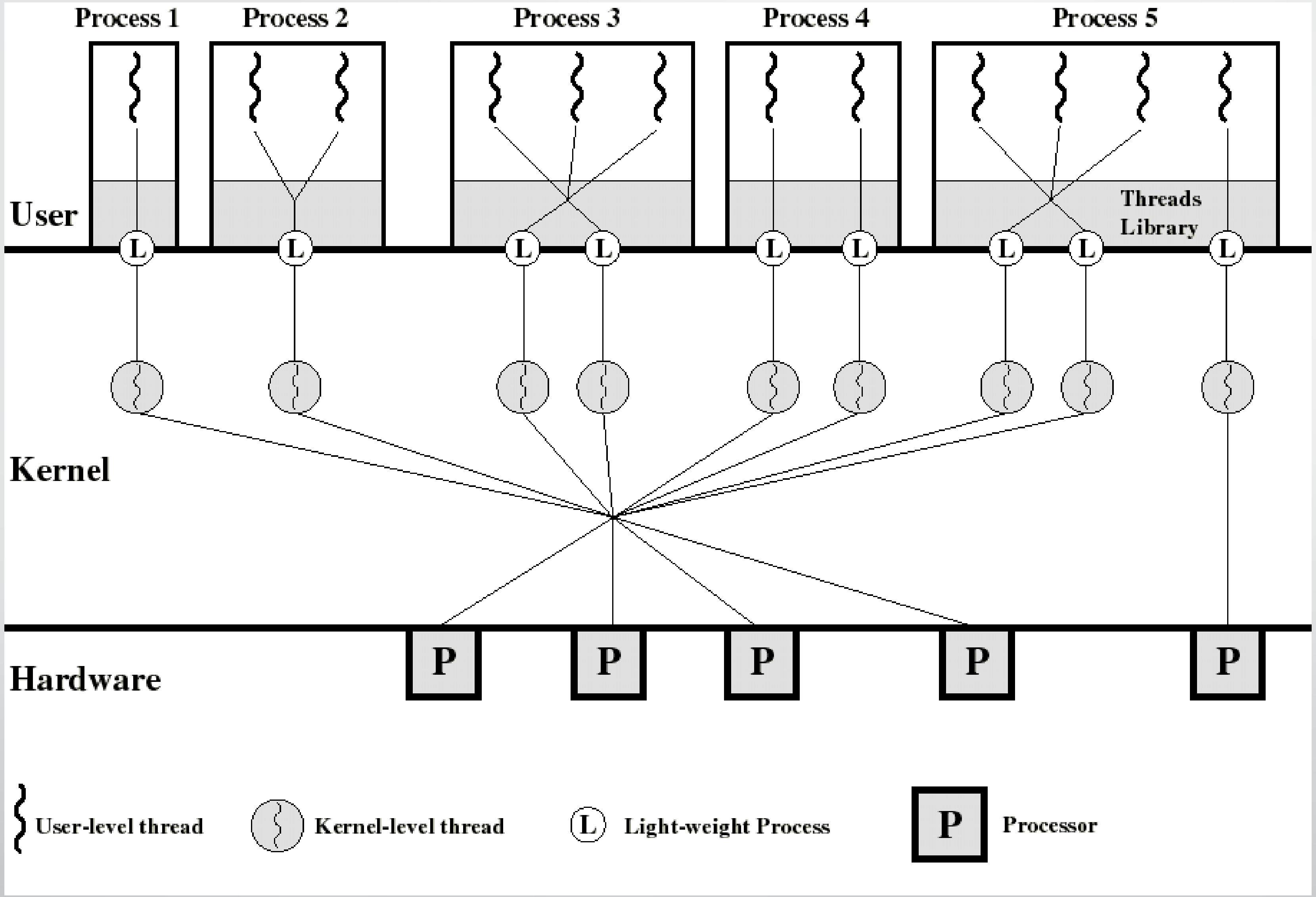
- Identifiers (real/effective user/group id)
- Timers (time spent in user/kernel mode)
- Signal Handler array
- Control terminal (if it exists)
- System call return value
- System call errors
- I/O and File parameters
- File Descriptor information
- Permission Mode Field
- Limit on process size



Solaris Process structure and traditional Unix Process structure

Threads

- Process: This is the normal UNIX process and includes the user’s address space, stack, and process control block.
- User-level threads: Implemented through a threads library in the address space of a process, these threads are invisible to the OS.A user-level thread (ULT)10 is a user-created unit of execution within a process.
- Lightweight processes: A lightweight process (LWP) can be viewed as a mapping between user level threads and kernel threads. Each LWP supports ULT and maps to one kernel thread. LWPs are scheduled by the kernel independently and may execute in parallel on multiprocessors.
- Kernel threads: These are the fundamental entities that can be scheduled and dispatched to run on one of the system processors.



Process 2 is equivalent to a pure ULT approach
Process 4 is equivalent to a pure KLT approach
We can specify a different degree of parallelism (process 3 and 5)

Solaris: versatility

- We can use ULTs when logical parallelism does not need to be supported by hardware parallelism (we save mode switching)

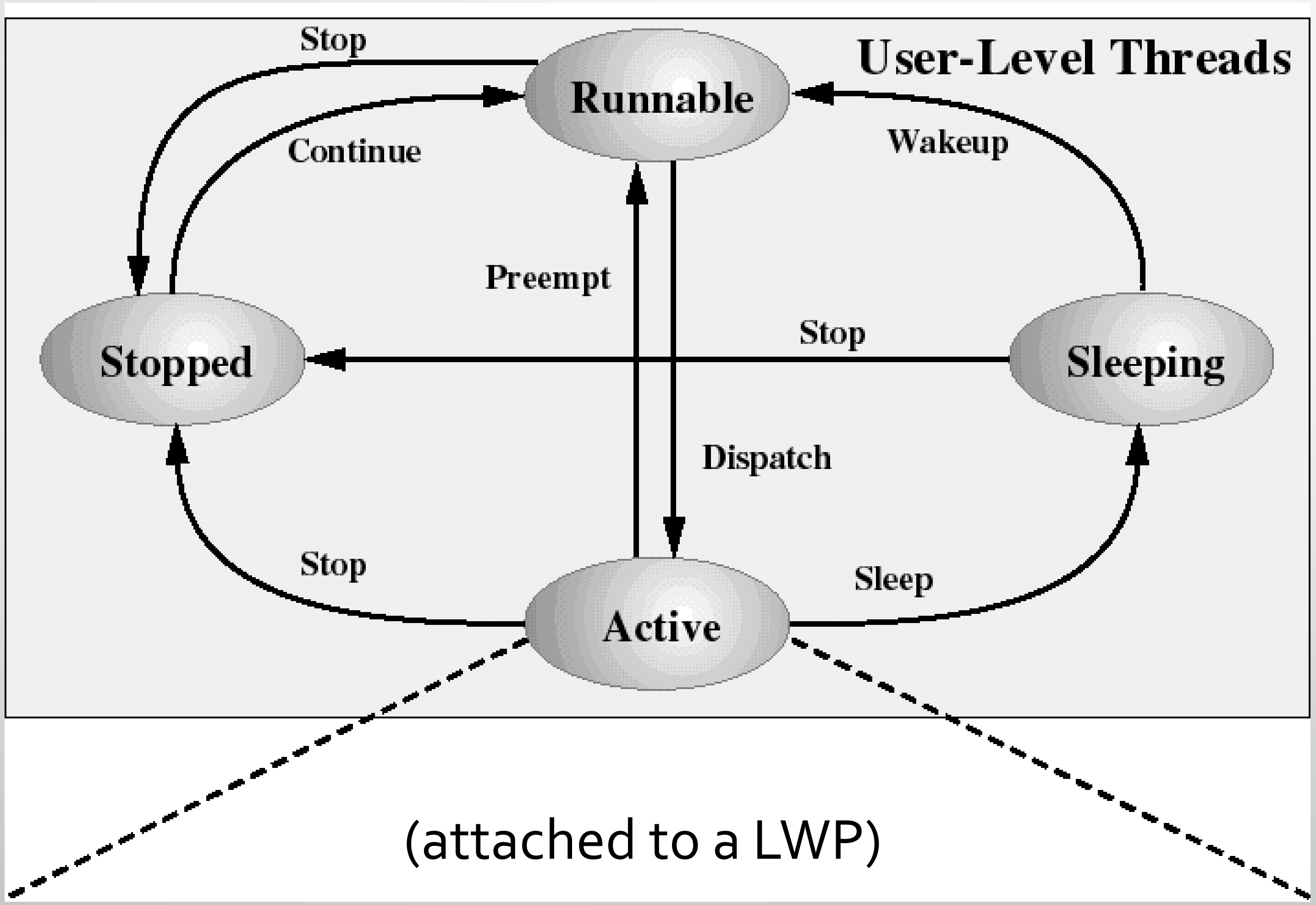
  Ex: Multiple windows but only one is active at any one time

- If threads may block then we can specify two or more LWPs to avoid blocking the whole application

user-level thread execution

- Transitions among these states is under the exclusive control of the application
  - a transition can occur only when a call is made to a function of the thread library
- It's only when a ULT is in the active state that it is attached to a LWP (so that it will run when the kernel level thread runs)
  - a thread may transfer to the sleeping state by invoking a synchronization primitive (chap 5) and later transfer to the runnable state when the event waited for occurs
  - A thread may force another thread to go to the stop state...

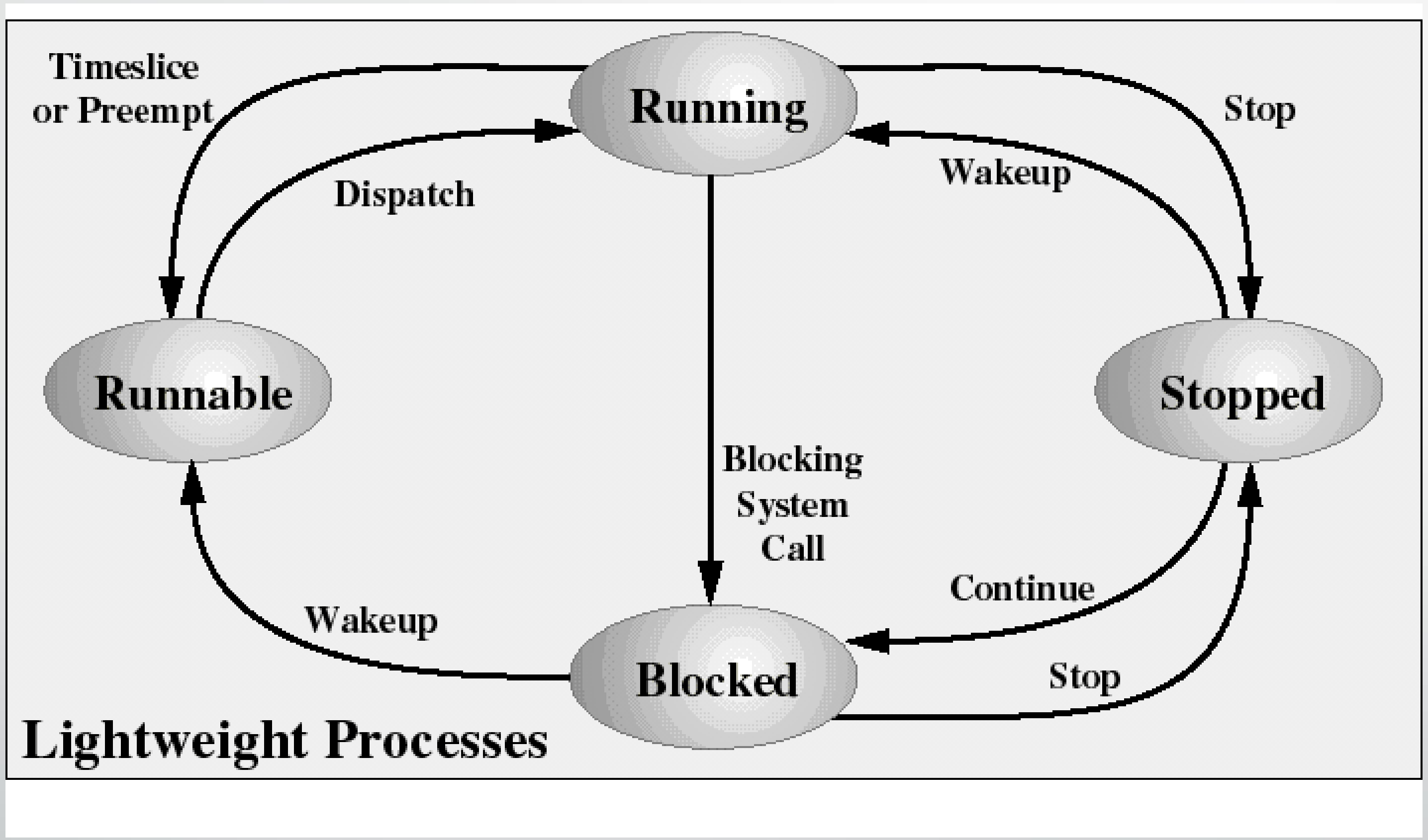
user-level thread states



Decomposition of user-level *Active* state

- When a ULT is Active, it is associated to a LWP and, thus, to a KLT
- Transitions among the LWP states is under the exclusive control of the kernel
- A LWP can be in the following states:
  - running: when the KLT is executing
  - blocked: because the KLT issued a blocking system call (but the ULT remains bound to that LWP and remains active)
  - runnable: waiting to be dispatched to CPU

Lightweight Process States



LWP states are independent of ULT states
(except for bound ULTs)

The Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Less time to switch between two threads within the same process
- Threads within the same process share memory and files --> they can communicate without invoking the kernel